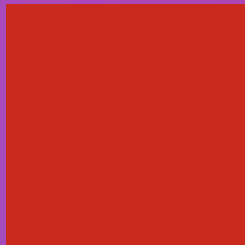


PROGRAMMIEREN LERNEN MIT

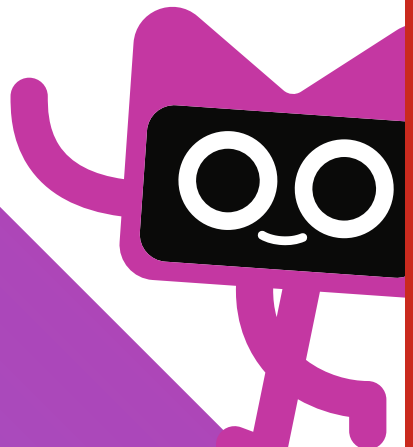
Kotlin

Grundlagen, Objektorientierung
und fortgeschrittene Konzepte

2. Auflage



Christian KOHLS
Alexander DOBRYNIN



Zusatzmaterial unter
plus.hanser-fachbuch.de

HANSER

Kohls/Dobrynin
Programmieren lernen mit Kotlin



Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf plus.hanser-fachbuch.de einfach diesen Code ein:

plus-rn34m-tL9pr



Blieben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter



Christian Kohls
Alexander Dobrynin

Programmieren lernen mit Kotlin

Grundlagen, Objektorientierung
und fortgeschrittene Konzepte

2., aktualisierte Ausgabe

HANSER

Die Autoren:

Prof. Dr. Christian Kohls, Köln

Alexander Dobrynin, Gummersbach

Kotlin ist ein eingetragenes Warenzeichen der Kotlin Foundation

Alle in diesem Werk enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Werk enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials – oder Teilen davon – entsteht. Ebensovienig übernehmen Autoren und Verlag die Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt also auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Die endgültige Entscheidung über die Eignung der Informationen für die vorgesehene Verwendung in einer bestimmten Anwendung liegt in der alleinigen Verantwortung des Nutzers.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung – mit Ausnahme der in den §§ 53, 54 URG genannten Sonderfälle –, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2023 Carl Hanser Verlag München, <http://www.hanser-fachbuch.de>

Lektorat: Sylvia Hasselbach

Copy editing: Jürgen Dubau, Freiburg/Elbe

Layout: le-tex publishing services GmbH

Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München

Titelmotiv und Umschlagrealisation: Max Kostopoulos

Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen

Printed in Germany

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

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

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

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

1

Einführung

Kotlin ist eine Programmiersprache, die verschiedene Ansätze verfolgt. Sie erlaubt objektorientierte und funktionale Programmierung. Das Entwickeln mit objektorientierten Programmiersprachen gehört heute zu den Grundanforderungen aller Informatikerinnen und Informatiker. Es gibt Ihnen die Möglichkeit, eigene Projektideen umzusetzen, betriebliche Prozesse zu modellieren und nützliche Werkzeuge für Anwender zu entwickeln. Die objektorientierte Herangehensweise schärft zudem Ihre analytischen und ganzheitlichen Denkfähigkeiten. Denn Sie müssen Sachverhalte differenziert betrachten, Theorien und Modelle entwickeln, Prozesse analysieren und optimieren, Probleme ganzheitlich betrachten und abwägen, Probleme in Teilprobleme zerlegen, die (Anwendungs-) Story verstehen und erzählen können. Zudem erlernen Sie ein systematisches Vorgehen für Problemlöseaufgaben und damit kreative Denkweisen. Objektorientierte Denker sind Weltversteher und Weltveränderer!

Weltversteher, weil sie die wahrgenommene Welt der Situation entsprechend und der Aufgabe angemessen in Klassen und Objekte, Zustandsbeschreibungen und Verhaltensweisen, Rollen und Beziehungen abbilden können. Weltveränderer, weil gute (und leider auch schlechte) Software unmittelbaren Einfluss auf den Lebensalltag der Anwender hat. Denken Sie nur daran, was mit Software alles möglich ist. Und bald werden Sie die nächste großartige App entwickeln! Während Objekte unsere Welt besonders gut abbilden, ist der Vorteil von Funktionen, dass sie Rechenregeln und Transformationen mit mathematischer Präzision festlegen können. Funktionale Programmieransätze bauen darauf auf und unterstützen uns dabei, Funktionalitäten miteinander zu verknüpfen, zu testen und wiederzuverwenden. Kotlin vereint diese und andere Konzepte zu einer modernen Sprache.

Kotlin ist eine noch recht junge Programmiersprache, die sich jedoch rasant durchsetzt. Fast alle kommerziell erfolgreichen Android-Apps werden derzeit in Kotlin entwickelt. Und das, obwohl Kotlin der Öffentlichkeit erst im Jahr 2011 erstmals vorgestellt wurde. Eine stabile Version gibt es seit 2016. Inzwischen hat Kotlin die Version 1.9 erreicht. Diese Version wurde am 6. Juli 2023 veröffentlicht, also kurz vor Redaktionsschluss dieses Buches. Wir haben die Neuerungen bereits in diesem Buch berücksichtigt und weisen an einzelnen Stellen darauf hin, wenn eine bestimmte Kotlin-Version benötigt wird. Das nächste größere Release wird Kotlin 2.0 sein.

Wir haben alle Kotlin-Entwicklerkonferenzen (KotlinConf) der letzten Jahre besucht und sind von der großen und aktiven Entwicklercommunity sehr begeistert. Neue Sprachfeatures werden mit der Community diskutiert, und Google bevorzugt Kotlin nicht nur als

Entwicklungssprache für die Android-Plattform, sondern stellt selbst große Teile des eigenen Codes auf Kotlin um. Viele bekannte Firmen setzen auf Kotlin und teilen auf der KotlinConf ihre Erfahrungen. Damit ist Kotlin im Gegensatz zu vielen anderen neuen Sprachen keine „akademische Sprache“, die nur für die Hochschullehre interessant ist. Ganz im Gegenteil: Immer mehr Firmen, Entwickler und Open-Source-Projekte wechseln zu Kotlin, da sich mit dieser Sprache Lösungen schneller und sicherer entwickeln lassen. Mit Jetpack Compose steht zudem ein sehr guter und moderner Ansatz für die Entwicklung grafischer Benutzeroberflächen bereit, mit dem sich nicht nur mobile Apps sehr schnell umsetzen lassen, sondern auch Desktop-Anwendungen für Ihren Mac oder PC.

Kotlin wurde von der Firma JetBrains entwickelt und steht unter einer Open Source-Lizenz. JetBrains ist ein führender Anbieter für integrierte Entwicklungsumgebungen wie z. B. IntelliJ. Bei JetBrains hat man irgendwann festgestellt, dass die Entwicklung mit der Programmiersprache Java an vielen Stellen zu aufwendig ist. Kotlin wurde basierend auf dem Erfahrungswissen, was effiziente Softwareentwickler wirklich benötigen, konzipiert. Kotlin vereint dabei viele erprobte Konzepte aus den letzten Jahrzehnten. Kotlin setzt konsequent auf Mechanismen, die sich bewährt haben und die in der Programmierpraxis zu einer erhöhten Produktivität führen – und dadurch letztlich mehr Freude bereitet. Ja, Kotlin macht Spaß!

■ 1.1 Eine Sprache für viele Plattformen

Mit Kotlin können Sie für viele verschiedene Plattformen entwickeln.

Android: Kotlin ist die primäre Entwicklungssprache für die Android-Plattform von Google. Damit hat sie eine hohe Relevanz, da sich Apps für Android mit Kotlin schneller, besser und leichter entwickeln lassen.

JVM: Programme, die in Kotlin geschrieben werden, können für die Java-Plattform kompiliert werden. Das heißt: Die Kotlin-Programme werden in Java-Bytecode übersetzt und laufen dann auf jeder Java Virtuellen Maschine (JVM). Diese JVMs gibt es für verschiedene Betriebssysteme. Ihre Programme können also auf verschiedenen Rechnersystemen ausgeführt werden.

JavaScript: Kotlin kann aber auch nach JavaScript übersetzt werden. Das heißt: Sie können die Programmierung von Webanwendungen auch mit Kotlin durchführen. Dies gilt sowohl für die Programmierung des Clients (also JavaScript, das im Browser ausgeführt wird) als auch des Servers (also JavaScript, das auf dem Server ausgeführt wird).

Kotlin Native: Darüber hinaus gibt es mit Kotlin Native den Ansatz, ein Kotlin-Programm direkt für eine bestimmte Rechnerarchitektur zu kompilieren. Es wird also nicht Bytecode für eine virtuelle Maschine wie die JVM erzeugt, sondern echter Maschinencode für die jeweilige Plattform (z. B. iOS, Mac oder Linux).

Kotlin Multiplattform: Die Multiplattform-Entwicklung zielt darauf ab, dass Sie Ihre Programme einmal schreiben und dann auf verschiedenen Plattformen wie z. B. Android, iOS, Desktop-Systeme und im Web laufen lassen können. Dabei wird die Anwendungslogik geteilt und für alle Systeme verwendet. Sie legen z. B. nur einmal fest, wie Daten verarbeitet

werden sollen. Die Gestaltung der Benutzeroberfläche kann dann die nativen Bedienelemente der verschiedenen Plattformen berücksichtigen. Mit Jetpack Compose, das wir ebenfalls in diesem Buch behandeln, ist es sogar möglich, die gesamte Benutzeroberfläche plattformübergreifend zu gestalten. Diese Technologie befindet sich allerdings aktuell noch zum Teil in der Alpha-Version (Stand Juli 2023).

■ 1.2 Deshalb ist Kotlin so besonders

Hier noch einmal die wichtigsten Gründe, die für Kotlin sprechen:

Einfacher Einstieg: Kotlin lässt sich schneller lernen. Es ist verständlicher, und die Sprachelemente sind prägnanter, also ausdrucksstärker. Sie können sich besser auf die wesentlichen Konzepte fokussieren, da unnötiger „Boilerplate Code“ entfällt. Das ist Code, der eigentlich nicht nötig ist, z. B. weil er keine neuen Sachverhalte ausdrückt. Ein Beispiel sind die vielen setter- und getter-Methoden, die man in anderen Programmiersprachen wie zum Beispiel Java schreiben muss.

Eleganter: Viele Programme lassen sich mit Kotlin eleganter schreiben. Mehr noch: Man muss die Programme eleganter schreiben. Sie lernen also gleich den richtigen Stil und können diesen auch in anderen Sprachen einsetzen.

Effiziente Entwicklung: Vieles lässt sich in Kotlin kürzer und effizienter ausdrücken. Das spart nicht nur Zeit beim Schreiben des Quellcodes. Durch kurze, übersichtliche Programme lassen sich Fehler besser vermeiden, und Sie behalten besser den Überblick.

Effiziente Ausführung: Kotlin stellt viele optimierte Algorithmen für immer wieder anfallende Aufgaben zur Verfügung. Das Verarbeiten oder Sortieren von Listen ist in Kotlin sehr gut gelöst und leicht nutzbar.

Erprobte Konzepte: Kotlin baut auf erprobten Konzepten auf. Die Sprache enthält konzeptionell nichts Neues, dafür (fast) alles Gute aus den letzten 50 Jahren Softwareentwicklung. Die Sprache ist praktisch und effizient, bietet mehr Sicherheit und vereint objektorientierte und funktionale Programmierkonzepte.

Erklärt sich selbst: Der Quellcode ist lesbarer für Menschen. Sie können den Code von anderen Entwicklern (und womöglich Ihren eigenen Code selbst nach ein paar Wochen) viel besser verstehen.

Error-less: Viele Fehler, die man in anderen Sprachen leicht machen kann, werden in Kotlin gar nicht erst zugelassen oder zumindest leichter vermieden. So gibt es in Kotlin beispielsweise keine sogenannten *Null-Fehler* mehr.

Erfrischend: Viele Entwickler berichten, dass das Entwickeln mit Kotlin wieder mehr Spaß macht! Und tatsächlich bekommt man beim Entwickeln mit Kotlin leuchtende Augen. Und manchmal auch tränende Augen – vor Freude über Kotlin und Ärger darüber, dass früher so vieles anstrengender war.

■ 1.3 Darauf dürfen Sie sich freuen

Aufbau des Computers: Programme laufen auf dem Prozessor eines Computers. Dabei verändern sie Daten im Speicher. Um zu verstehen, wie aus dem Quellcode, den Sie schreiben, lauffähige Programme auf dem Rechner werden, schauen wir uns zunächst den grundlegenden Aufbau von Computern an. Dabei werden wir sehen, wie aus lauter Nullen und Einsen bunte Bilder werden können. Da wir Daten im Speicher liegen haben, werden wir uns auch mit der Organisation des Speichers beschäftigen.

Basics: Zur Steuerung von Programmen benötigen wir ein paar grundlegende Zutaten. Dazu gehören Ausdrücke und Anweisungen, um dem Rechner zu sagen, was er ausführen soll. Mit Variablen können wir Daten speichern und später wieder darauf zugreifen. Mit Kontrollstrukturen können wir steuern, wie ein Programm abläuft. Somit können wir auf Benutzereingaben und verschiedene Zustände reagieren. Mit Funktionen werden wir wiederverwendbare Codebausteine definieren, aus denen sich komplexe Programme zusammensetzen lassen.

Objekte und Klassen: Wir werden Grundlagen über Objekte und Abstraktion als Klassen kennenlernen. Wie können wir ermitteln, welche Eigenschaften von Objekten relevant sind? Wie abstrahiere ich von konkreten Objekten in der Welt auf eine allgemeine Klasse? Klassen beschreiben die Struktur, die Eigenschaften und die Verhaltensweisen von allen Objekten, die zu einer Klasse gehören. Man spricht von Objektinstanzen (oder einfache Instanzen), wenn man sich auf die Exemplare einer Klasse bezieht. Das Erzeugen eines neuen Objekts erfolgt über Konstruktoren. Wir werden uns ansehen, wie Konstruktoren die erste Version eines Objekts bauen. Zudem werden wir Objekte miteinander in Beziehung setzen. Zum Beispiel werden wir einen Film aus Budget, Schauspieler und Regisseur zusammensetzen, um dies in einem Spiel zu verwenden.

Vererbung und Polymorphie Dies ist ein weiteres zentrales Thema der objektorientierten Programmierung. Klassen stehen in einer Vererbungshierarchie zueinander. Klassen können andere Klassen erweitern, wobei die Eigenschaften und Methoden einer Oberklasse an die Unterklasse vererbt werden. Wenn die übergeordnete Klasse Tasse die Eigenschaft „Volumen“ und die Methode „trinken“ besitzt, dann wird auch die untergeordnete Klasse Kaffeetasse diese Eigenschaften haben, sie kann aber um spezifische Eigenschaften (z. B. „Kaffeesorte“) und Methoden (z. B. „istNochHeiss“) ergänzt werden. Dabei kann das Verhalten von Methoden auch durch Überschreiben geändert werden (z. B. weil man aus einer Thermotasse anders trinkt). Eng verbunden mit der Definition von Klassen und Schnittstellen sind Typkonzepte und Polymorphie. Polymorphie bedeutet Vielgestaltigkeit. Wenn Sie an „Fahrzeug“ denken, dann kann dieses Fahrzeug sehr viele unterschiedliche Gestalten haben: Auto, Fahrrad, Kutsche. Dennoch können Sie allgemeine Eigenschaften und Methoden nutzen, wenn diese Gestalten auf einer abstrakteren Ebene vom gleichen Typ sind. So lässt sich für alle Fahrzeuge sagen: „fahre los“. Egal, ob es sich um eine Kutsche oder ein Fahrrad handelt. Die Umsetzung wird aber ganz unterschiedlich aussehen.

Robustheit: Damit wir auch ordentliche Programme schreiben können, werden wir uns mit verschiedenen Konzepten zur Robustheit von Code beschäftigen. Wie können Sie auf Fehler und Ausnahmesituationen reagieren? Was passiert, wenn eine Datei gelesen werden soll, die gar nicht existiert? Was passiert, wenn ein Server nicht erreichbar ist? Ausnahmezustand! Wir werden lernen, wie wir eine Operation versuchen können und das Scheitern schon

einplanen und auffangen. Das funktioniert bestens für Situationen, wo Sie selbst keinen Einfluss darauf haben, ob etwas wie gewünscht funktioniert. Dann können Sie nämlich darauf reagieren. Denn das Auffangen ist viel komplizierter als die Fehlerkorrektur. Und vor allem wollen Sie bei der Entwicklung ja auch, dass Fehler erkannt und somit beseitigt werden können. Wir werden uns die Vor- und Nachteile für verschiedene Software-Designoptionen anschauen. Aus diesen Überlegungen heraus haben sich über die Jahre hinweg Entwurfsmuster entwickelt, die gute Lösungen für bestimmte Aufgaben darstellen.

Datensammlungen: Wir schauen uns verschiedene Möglichkeiten an, um Datensammlungen (z. B. Listen, Verzeichnisse, Paare, Mengen) abzubilden und Operationen darauf auszuführen. Zudem werden wir selbst Datenstrukturen entwickeln, um Objekte zu speichern. Dies wird unter anderem am Beispiel einer verketteten Liste illustriert.

UI Design und App-Entwicklung: Die Android-Plattform stellt spezielle Bibliotheken zur Verfügung, um grafische Oberflächen zu gestalten und die Funktionen von Android-Geräten zu nutzen. Wir werden eine grafische Oberfläche bauen und Code definieren, um auf Eingabeevents (Klick auf einen Button) zu reagieren.

Nebenläufigkeit: Häufig müssen mehrere Dinge gleichzeitig ausgeführt werden. Bei Spielen sollen zum Beispiel gleichzeitig mehrere Figuren bewegt werden. Auf Ihrem Smartphone sollen gleichzeitig mehrere Bilder aus dem Internet heruntergeladen werden, während Sie weiterhin mit dem Programm interagieren. Für diese Nebenläufigkeit führt Kotlin das Konzept der *Coroutines* ein. Wir werden uns anschauen, wie sich nebenläufige Programme gestalten lassen und auf welche Stolpersteine geachtet werden muss.



TEIL I

Konzeptioneller Aufbau von Computern und Software

Um ein besseres Verständnis zu erlangen, was es eigentlich heißt, Software zu entwickeln und lauffähige Programme zu schreiben, müssen wir uns ein wenig mit der Hardware beschäftigen – dem Computer. Computer sind heute allgegenwärtig. Damit sind nicht nur die großen Kisten unterm Schreibtisch oder die mobilen Laptops gemeint. Auch Ihr Smartphone ist ein Computer! Und zwar ein Computer, der sehr viel leistungsfähiger ist als die großen Geräte, die noch vor ein paar Jahren unter dem Schreibtisch standen. Darüber hinaus erhalten kleine Computer jedoch auch Einzug in immer mehr Alltagsgegenstände und Maschinen: von der Zahnbürste über die Waschmaschine, über den Fahrkartenautomat hin zum Autopiloten in Bahnen oder Flugzeugen und natürlich der Bordcomputer in unseren Autos. Viele Waschmaschinen nutzen inzwischen dieselben Chips, die einst in unseren ersten Heimcomputern eingebaut wurden.

Doch was ist eigentlich ein Computer und woraus setzt er sich zusammen? Der Begriff *Computer* leitet sich aus dem englischen Verb *compute* ab, welches mit *rechnen* übersetzt werden kann. Daher wird im deutschsprachigen Raum der Computer auch häufig als *Rechner* bezeichnet. Tatsächlich macht ein Computer nichts anderes: Er rechnet und rechnet und rechnet und rechnet.

Das klingt vielleicht etwas merkwürdig, wenn wir an die uns vertrauten Anwendungen denken: Was hat die Kurznachricht bei WhatsApp oder Twitter mit Berechnungen zu tun? Wo bitte schön sind die Berechnungen bei grafikintensiven Spielen wie SimCity, Angry Birds oder Fortnite? Die Antwort lautet: ziemlich versteckt, aber dafür überall. Und daher wollen wir uns in diesem Kapitel ein kleines bisschen auf die Suche nach diesen Berechnungen und Zahlen machen. Denn wenn wir verstehen, was unsere Software und Hardware im Innersten zusammenhält, dann bekommen wir auch ein besseres Verständnis für die Funktionsweise der Programme, die wir selbst entwickeln. Und wir werden verstehen, was eigentlich ein „Programm“ ist!

2

Komponenten eines Computers

Zunächst besteht ein Computer nur aus diesen wesentlichen Komponenten:

- Speicher, in dem Daten liegen
- Prozessoren, die diese Daten verarbeiten
- Eingabe- und Ausgabegeräte

Die Realität sieht natürlich sehr viel komplexer aus, denn es gibt verschiedene Speicherarten, auf die unterschiedlich schnell zugegriffen werden kann. Zudem kann Speicher unterschiedlich organisiert werden – ganz so, wie Sie auch Ihre Wäsche unterschiedlich im Schrank anordnen können. Dabei gibt es für verschiedene Zwecke optimierte Vorgehensweisen. Und es gibt unterschiedliche Prozessoren, die für die Verarbeitung bestimmter Daten optimiert sind. So gibt es etwa Prozessoren, die allgemeine Rechenoperationen ausführen können, und spezialisierte Prozessoren, die besonders schnell Berechnungen für die Grafikausgabe oder die Sounderzeugung anstellen können. Zudem müssen die Daten aus den Speichern zu den Prozessoren gelangen, und auch die Ein- und Ausgabegeräte wollen natürlich etwas mit den Daten anfangen. Um diese Daten zu befördern, gibt es sogenannte Datenbusse. Auch diese können unterschiedlich schnell sein. Auch bei den Ein- und Ausgabegeräten gibt es viele verschiedene, Ihnen bekannte Geräte. Typische Eingabegeräte sind die Tastatur, eine Maus, eine touchfähige Oberfläche über dem Bildschirm, Sensoren für Temperatur, Bewegung oder Helligkeit, Scanner, Kameras und viele mehr. Zu den typischen Ausgabegeräten gehören Bildschirme, Lautsprecher, Drucker, aber z. B. auch Roboter, die durch Computer gesteuert werden.

■ 2.1 Beliebige Daten als binäre Zahlen

Und wenn wir von Daten reden, dann sind diese intern stets und ohne Ausnahme als Zahlen repräsentiert. Ja, jede Kurznachricht, jedes Meme, jedes Video, jeder Text und jede Fahrbewegung eines Roboters, Autos oder Flugzeugs wird intern durch Zahlen abgebildet. Genauer gesagt durch Binärzahlen, also Zahlen, die nur aus 0 und 1 bestehen. Dies hat einen guten Grund: Zwei Zustände lassen sich sehr gut durch physische Hardware abbilden, z. B. durch unterschiedliche magnetische Ausrichtungen oder Spannungen. Mit diesen zwei voneinander unterscheidbaren Zuständen lassen sich prinzipiell alle Arten von Daten und Informationen abbilden, indem sie unterschiedlich interpretiert werden:

Physischer Zustand	Interpretation als Schaltung	Interpretation als Antwort	Interpretation als Wahrheitsaussage	Interpretation als Zahl
Oberhalb Grenzwert	EIN	Ja	WAHR	1
Unterhalb Grenzwert	AUS	Nein	FALSCH	0

Bei der Interpretation als Zahl können wir mehrstellige Zahlen abbilden, wenn mehrere Bits gemeinsam betrachtet werden. Dies ist vergleichbar mit dem uns vertrauten Dezimalsystem, also dem Zahlensystem mit den Ziffern 0–9. Während dieses Dezimalsystem aus nur 10 verschiedene Ziffern besteht, können wir dennoch beliebig große Zahlen abbilden, z. B.:

> 342359

Die Position der Ziffer gibt dabei die Wertigkeit an. So bedeutet die 9 an letzter Position, dass es sich um den Wert 9 handelt. Die Ziffer an der vorletzten Position bedeutet dagegen, dass sie mit dem Faktor 10 multipliziert wird. Die fünf bedeutet also 50. Genauer gesagt wird die Ziffer an der Position n (von hinten ausgehend) mit dem Faktor 10^n multipliziert. Die letzte Position ist dabei die 0. Stelle, sodass für die Beispielzahl gilt:

Ziffer	Konvertierung	Wertigkeit als Dezimalzahl	Summe
3	$\times 10^5$	100000	300000
4	$\times 10^4$	10000	40000
2	$\times 10^3$	1000	2000
3	$\times 10^2$	100	300
5	$\times 10^1$	10	50
9	$\times 10^0$	1	9
Summe:			342359

Während sich beim Dezimalsystem die Wertigkeit bei jeder Position verzehnfacht (also Faktor 10), verdoppelt sich beim Binärsystem die Wertigkeit bei jeder Position (also Faktor 2). Auf diese Weise lässt sich eine Binärzahl mit mehreren Ziffern zusammensetzen. In Computern werden dabei jeweils mindestens 8 Stellen zu einer Einheit, die *Byte* genannt wird, zusammengefasst. Eine Binärzahl ist z. B.:

> 00010101

Wenn man diese Binärzahl in eine Dezimalzahl umrechnen möchte, dann multipliziert man jede Stelle mit 2^n statt mit 10^n . So ergibt sich für das gerade angeschaute Beispiel: