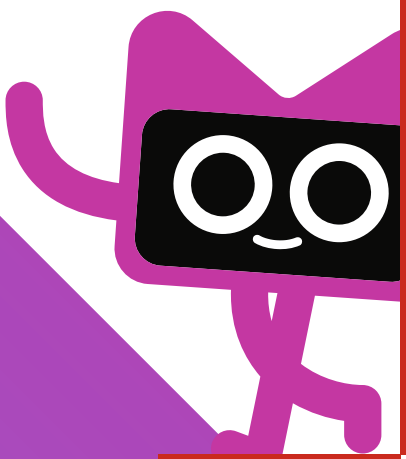
PROGRAMMIEREN LERNEN MIT

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

Bleiben 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

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.

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

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 Wahrheits-aussage	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	x 10^5	100000	300000
4	x 10^4	10000	40000
2	x 10^3	1000	2000
3	x 10^2	100	300
5	x 10^1	10	50
9	x 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:

Ziffer	Konvertierung	Wertigkeit als Dezimalzahl	Summe
0	x 2^7	128	0
0	x 2^6	64	0
0	x 2^5	32	0
1	x 2^4	16	16
0	x 2^3	8	0
1	x 2^2	4	4
0	x 2^1	2	0
1	x 2^0	1	1
Summe:			21

Die größte Zahl, die sich mit 8 Bits darstellen lässt ist die Zahl 11111111. Umgerechnet ist dies die Zahl 255:

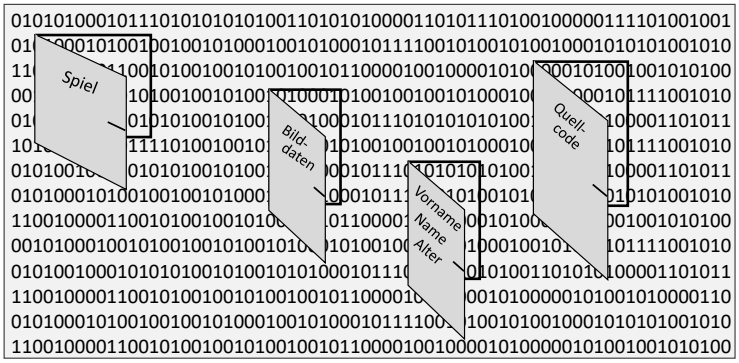
Ziffer	Konvertierung	Wertigkeit als Dezimalzahl	Summe
1	x 2^7	128	128
1	x 2^6	64	64
1	x 2^5	32	32
1	x 2^4	16	16
1	x 2^3	8	8
1	x 2^2	4	4
1	x 2^1	2	2
1	x 2^0	1	1
Summe:			255

Die kleinste Zahl ist dagegen die 00000000. Sie entspricht auch der 0 in unserem Dezimalsystem. Insgesamt können mit 8 Bits bzw. 1 Byte also 256 verschiedene Zahlenwerte abgebildet werden, nämlich alle Dezimalzahlen zwischen 0 und 255. Weil die 0 die erste Zahl ist, werden wir an vielen Stellen sehen, dass wir in der Informatik und bei der Programmierung häufig bei 0 anfangen zu zählen. Gerade zu Beginn der Computerzeit war Speicher eine sehr knappe Ressource, und man versuchte immer, den gesamten verfügbaren Wertebereich auszuschöpfen.

256 verschiedene Werte erscheinen erst einmal recht wenig. Aber sobald man 2 Bytes verwendet, hat man 256 x 256 verschiedene Wertemöglichkeiten, genauer gesagt 65536 verschiedene Möglichkeiten. In Kotlin werden für die Speicherung von ganzen Zahlen in der Regel 4 Bytes verwendet, sodass 4294967296 zur Verfügung stehen. Damit auch negative Werte abgebildet werden können, wird die erste Hälfte davon als negative Zahl interpretiert, die zweite Hälfte als positive Zahl. Der Wertebereich liegt also zwischen -2147483648 und 2147483647.

2.2 Wie Zahlen in Texte, Bilder und Animationen umgewandelt werden

Schön, jetzt wissen wir schon einmal, wie aus physischen Zuständen der Hardware Zahlen werden können. Doch wie werden aus Zahlen denn nun Texte in einer Kurznachricht, Songs auf dem Smartphone oder Bilder in einem Spiel? Die Antwort lautet: Wenn wir die Nullen und Einsen als Werte für Dezimalzahl umrechnen können, dann können wir sie auch noch auf andere Weise interpretieren. Eine bestimmte Reihenfolge von Zahlen kann z. B. als Code für einen Buchstaben oder als Farbe für einen Punkt auf dem Bildschirm interpretiert werden. Zum Beispiel lässt sich eine Farbe durch seinen roten, blauen und grünen Lichtanteil eindeutig beschreiben. Wenn wir für jeden Lichtanteil 256 Abstufungen einführen, dann ergeben sich 256 x 256 x 256 = 16.777.216 verschiedene Werte für verschiedene Farben. Dies wird häufig aufgerundet, und für Grafikkarten wird angegeben, dass 16,8 Millionen verschiedene Farben möglich sind. Man bezeichnet diese Farbwerte auch als RGB-Werte, basierend auf den Namen für die Farben rot (red), grün (green) und blau (blue). Wenn Sie nun eine ganze Reihe solcher Zahlen speichern, dann können Sie die Farbwerte für jeden Bildpunkt auf Ihrem Bildschirm, also die einzelnen Pixel, festlegen. Tatsächlich ist der bunte Bildschirm, den Sie auf dem Laptop oder Smartphone sehen, intern nichts anderes als eine sehr lange Aneinanderreihung vieler, vieler Zahlen, die für jeden Pixel einen RGB-Wert festlegen.



Wenn sich die Bildschirminhalte verändern, wenn z. B. in einem Videospiel ein zorniger Vogel durch die Gegend fliegt, dann geschieht intern nichts anderes als eine Berechnung, welche neuen Zahlenwerte an welchen Stellen geändert werden müssen. Es werden also Positionen innerhalb der Bildschirmmatrix und neue Farbwerte berechnet. Und hier sehen wir schon, dass der Begriff „Rechner“ oder „Computer“ genau richtig ist. Der Prozessor führt die Berechnungen aus, er verarbeitet die Daten aus dem Speicher und verändert den Speicher, um die berechneten Daten (z. B. neue Pixelfarben) dort hineinzuschreiben. Weil gerade für die Grafikwiedergabe sehr viele und sehr spezielle Berechnungen nötig sind, gibt es hierfür spezielle Grafikprozessoren, die GPUs (Graphic Processing Units). Neben Bildpixeln spielt aber natürlich auch Text eine besondere Rolle. Denn wir schreiben oft lange Texte am Rechner. Eine wichtige Textsorte, um die es hier in diesem Buch gehen wird, ist der Quelltext für ein Programm. Dabei handelt es sich um einen Text, der vom Computer als

ausführbares Programm interpretiert oder in ein solches übersetzt werden kann. Damit der Computer jedoch unsere Texte versteht, müssen wir uns an ein paar Regeln halten, die wir uns noch genauer anschauen.

Ähnlich wie bei den Farbcodes, bei denen Zahlen festlegen, in welcher Farbe ein Bild erstrahlen soll, können wir Zahlen bestimmten Buchstaben zuordnen. Damit dies einheitlich geschieht, also wir uns nicht jedes Mal ausdenken müssen, für welchen Buchstaben eine Zahl steht, gibt es internationale Standards. Zwei Standards sind für uns von Bedeutung. Zum einen gibt es den ASCII-Standard. ASCII steht für „American Standard Code for Information Interchange“ und legt für 128 Werte unterschiedliche Buchstaben, Ziffern und Sonderzeichen fest, z. B.:

Wert	Buchstabe
65	A
66	B
67	C
…	…
97	a
98	b
…	..

Wenn man nun eine Zahlensequenz hat, z. B. 72, 97, 108,108, 111, dann lässt sich jeder Zahl ein Buchstabe dieser ASCII-Tabelle zuordnen. In diesem Beispiel wären dies die Buchstaben H,a,l,l,o. Und aus diesen Buchstaben lässt sich dann das Wort „Hallo“ zusammensetzen.

Von den 128 ASCII-Codes sind 95 Codes für druckbare Zeichen vorgesehen, die restlichen Codes sind Steuerzeichen, z. B. das Zeichen für eine neue Zeile. Nun ist es allerdings so, dass viele Sprachen nicht mit 95 verschiedenen Zeichen auskommen, da es viele Sonderzeichen oder spezielle Schriftzeichen gibt. Daher gibt es mit Unicode einen weiteren Standard bzw. mehrere Standards, die einen umfangreicheren Zeichenvorrat besitzen. Für Kotlin kommt der UTF-16-Standard zum Einsatz, bei dem jedes Zeichen durch eine 16-Bit-Zahl repräsentiert wird und entsprechend viel mehr verschiedene Zeichen codiert werden können.

Die Nullen und Einsen, die im Speicher liegen, können also auf unterschiedlichste Weise interpretiert werden: als Dezimalzahlen, Farbwerte, Buchstaben, Werte für Audiosignale usw.

2.3 Zahlen als ausführbarer Code

Jedem Zahlencode kommt dabei eine besondere Bedeutung zu. Und dies können wir uns zunutze machen, um ausführbare Programme für Computer festzulegen. Denn wenn eine bestimmte Zahl, je nach Interpretation, einmal die Farbe „himmelblau“ und ein anderes Mal den Buchstaben „Z“ meinen kann, so lässt sich über Zahlencodes auch festlegen, was der Prozessor als Nächstes machen soll. Zum Beispiel könnten wir folgende Zahlenfolge auf unterschiedliche Weise interpretieren:

- 0 136 255 -> R, G, B Werte für himmelblau
- 0 136 255 -> bestimmte Buchstaben laut ASCII-Tabelle
- 0 136 255 -> Addiere 136 und 255

In der letzten Interpretation der drei Zahlen werden diese als ausführbarer Code aufgefasst. Dies kann uns gelingen, indem wir der Zahl 0 hier die Bedeutung der Addition zukommen lassen. 0 ist also unser **Code für die Addition** zweier Zahlen, die hier ebenfalls festgelegt sind. So wie eine Tabelle bestimmte Buchstaben einer Zahl zuordnet, können andere Tabellen bestimmte Befehle wie Addition oder Subtraktion einer Zahl zuordnen. Wir können also über Zahlen Befehle oder Instruktionen festlegen, die ein Prozessor interpretieren und ausführen kann. Ausführen bedeutet in diesem Fall, dass eine bestimmte Form der Berechnung (hier die Addition) mit bestimmten Daten (hier die Zahlen 136 und 255) ausgeführt werden soll.

In gleicher Weise können wir uns weitere Codes für andere Anweisungen überlegen, z. B. Multiplikation und Division, Zahlen vergleichen, Bildpixel verändern usw. Bei diesen Operationen (also bestimmte Arten der Berechnung) gibt es Operanden (also Datenwerte), auf denen die Operation ausgeführt und ein Ergebnis berechnet wird. Es gibt auch Anweisungen, die eine Reihe von Bits direkt verändern, z. B. alles auf 0 oder 1 setzen, die Positionen nach links oder rechts verschieben (shiften) oder zwei Bit-Reihen miteinander durch eine logische Operation verknüpfen. Zum Beispiel gibt es die ODER-Verknüpfung. Dabei wird an der jeweiligen Position als Ergebnis der Bitwert auf 1 gesetzt, wenn von den beiden Operanden mindestens einer an der entsprechenden Position auch eine 1 stehen hat. Bei der UND-Verknüpfung gilt hingegen, dass das Ergebnis nur dann 1 ist, wenn beide Operanden an der entsprechenden Bitstelle auch eine 1 stehen haben. Mit dieser Logik werden wir uns noch ausführlicher beschäftigen.

Für den Moment ist es erst einmal wichtig zu verstehen, dass Instruktionen für den Prozessor auch als Zahlen festgelegt werden können. Dadurch ist es möglich, ein Programm als Reihe von Zahlen festzulegen, die wiederum selbst im Speicher liegen. Dies ist das universelle Rechnerkonzept, wie es John von Neumann beschrieben hat. Instruktionen können nach und nach abgearbeitet werden. Welche Instruktion als Nächstes ausgeführt wird, kann wiederum als eine Zahl festgelegt werden (z. B. Führe Instruktion 44 aus, Führe Instruktion 45 aus, Führe Instruktion 46 aus …). Diese Zahl nennt man Programmzähler. Er gibt an, die wievielte Instruktion als Nächstes ausgeführt werden soll. Es gibt auch Instruktionen, um diesen Programmzähler zu verändern. Wenn z. B. eine Instruktionsfolge mehrfach ausgeführt werden soll, dann kann der Programmzähler auch wieder auf eine andere Zahl gesetzt werden (Führe wieder die Instruktion 44 aus). Solche wiederholten Abläufe nennt man Schleifen, und wir werden noch sehen, wie man selbst Schleifen in einem Programm festlegt. Der Programmzähler kann auch Sprünge machen, wenn z. B. Entscheidungen anstehen, ob eine bestimmte Sequenz von Instruktionen überhaupt ausgeführt werden soll: Nur wenn ich genug Geld habe, dann kaufe ich mir ein Eis. Die notwendigen Instruktionen, um „Eis kaufen“ als Programm abzubilden, sollen nur dann ausgeführt werden, wenn ich genug Geld habe. Wenn dies nicht der Fall ist, muss der Programmzähler an eine andere Stelle springen, um diesen bedingten Ablauf zu überspringen. Um festzustellen, ob ich genug Geld habe, kommen wiederum spezielle Instruktionen zum Einsatz, mit denen Werte verglichen und – je nach Ergebnis – der Programmzähler verändert wird.

3 Zugriff auf den Speicher

Beim Ausführen einer Operation ist es in der Regel wichtig, auf bereits existierende Daten im Speicher zuzugreifen und das Ergebnis auch wieder zurück in den Speicher zu schreiben. Der Speicher besteht im Prinzip aus einer sehr großen Anzahl von Speicherzellen. Jede einzelne Speicherzelle kann, wie bereits erwähnt, nur zwischen den Werten 0 und 1 unterscheiden. Erst durch den Zusammenschluss mehrerer Speicherzellen ergeben sich interessante Einheiten von Zahlenwerten, die dann wiederum auf unterschiedliche Weise interpretiert werden können. Damit auf einen solchen Zusammenschluss von Speicherzellen (z. B. 8 Bits als 1 Byte) zugegriffen werden kann, müssen diese Einheiten adressiert werden. Daher spricht man auch von Speicheradressen. Eine Speicheradresse besteht dabei nicht etwa aus „Blumenstraße 99“, sondern einfach aus einer nüchternen Zahl. Diese Adresszahl sagt z. B.: Verwende die Daten aus den Speicheradressen 390 und 506. Schreibe dann das Ergebnis in die Speicheradresse 7002.

Man kann sich dies so vorstellen, als würde man eine Reihe von Türen über die vielen Speicherzellen legen. Jede Tür hat eine eigene Adresse. Öffnet man die Tür an der Stelle 390, dann können dort andere Bitfolgen stehen als hinter der Tür 506.

Wie die Bitfolgen hinter den Türen interpretiert werden sollen, hängt von der Instruktion ab, die mit diesen Daten arbeiten wird. Dieselbe Bitfolge kann als Zahl, Farbwert, Buchstabe oder sogar als weitere Instruktion interpretiert werden.

Daher legt man bei der Verwendung der Daten, also quasi beim Öffnen der Türen über eine Adresse, auch fest, was mit den Daten geschehen soll. Dies geschieht über die Festlegung eines *Datentyps*. Wenn eine Bitfolge als Zeichen interpretiert wird, dann ist dies ein anderer Datentyp als bei der Interpretation als ganze Zahl. Der Typ legt dabei nicht nur fest, wie die Nullen und Einsen zu interpretieren sind, sondern auch, welche Arten der Veränderungen erlaubt sind. Anders gesagt: Der Typ bestimmt die **Interpretation der Datenwerte** und die **Anzahl der erlaubten Operationen**. Welche Typen uns von Kotlin angeboten werden, werden wir in Kapitel 9, „Basis-Datentypen“, kennenlernen. Wie wir eigene Typen festlegen, werden wir in Teil III, „Objektorientierte Programmierung“, noch sehen.

Übrigens:

Die unterschiedliche Interpretation der Zahlen als Werte in einem Programm oder als Instruktionen für ein Programm ist ein mächtiges Werkzeug. Es ermöglicht uns, im selben Speicher sowohl Daten als auch Programme zu speichern. Gleichzeitig ist es

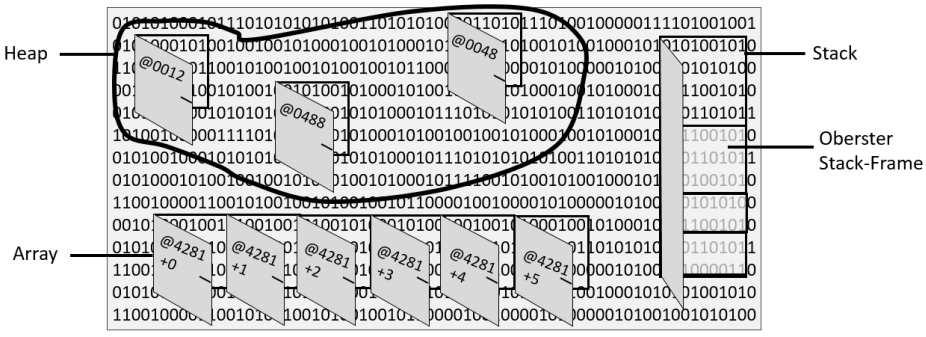
die Angriffsfläche, die leider auch Viren nutzen. Denn wenn es gelingt, die Zahlen im Speicher zu verändern, die als Instruktionen interpretiert werden, dann können Instruktionsfolgen eingeschleust werden, die schadhaft sind. So kann eine harmlos aussehende Operation, bei der ein Rechenergebnis in eine Speicherzelle geschrieben wird, in Wirklichkeit dazu führen, dass diese Zahl als Instruktion interpretiert wird, die schädliche Konsequenzen hat.

3.1 Organisation des Speichers

Es kann durchaus sein, dass für bestimmte Datentypen etwas mehr Speicherplatz benötigt wird. Nehmen wir an, dass für eine Person der Vor- und Nachname sowie das Alter gespeichert werden sollen. Dann lässt sich hierfür ein etwas größeres Bitfeld reservieren. Dies entspricht in unserer Metapher einer größeren Tür. Auch diese Tür hat eine Adresse, über die dann aber strukturiert auf Unterelemente zugegriffen werden kann.

Der Zugriff auf alle Türen im Hauptspeicher geschieht gleich schnell. Der Prozessor muss nur die Adresse wissen. Danach hat er einen direkten Zugriff auf die Bits hinter dieser Tür. Dieser Hauptspeicher heißt daher auch RAM (*Random Access Memory*). Der Zugriff kann auf beliebige Werte geschehen.

Neben dem Zusammenschluss mehrerer Einzeldaten hinter einer etwas größeren Tür ist es auch üblich, dass mehrere Daten desselben Typs zu einem Datenfeld (einem Array) zusammengefasst werden. Dies entspricht im Prinzip einer Häuserfront mit einer fixen Anzahl von Türen, hinter denen Daten desselben Typs gespeichert werden. Für eine Zeichenkette wird z. B. ein Feld mit Buchstaben benötigt. Damit diese zusammengehörenden Elemente nicht wild verstreut sind und man nach dem jeweils nächsten Buchstaben suchen muss, speichert man diese hintereinander. Dadurch lässt sich auch sehr schnell auf die einzelnen Türen des Feldes zugreifen: Wenn Sie wissen, dass Laura in der Straße 23 hinter der 4. Tür wohnt, dann kommen Sie schnell dort hin. Wenn Laura aber etwas flexibler ist und mal hier, mal dort wohnt, dann müssen Sie vielleicht bei mehreren Türen nachschauen, um Laura zu finden. Arrays ermöglichen also einen schnellen Zugriff, sind jedoch etwas weniger flexibel.



3.2 Daten im Speicher und Datenverarbeitung im Prozessor

Während der Zugriff gezielt über Speicheradressen erfolgt, müssen diese noch zum Prozessor gelangen, um dort verarbeitet zu werden. Die Daten müssen also für die Verarbeitung ausgelesen und das Ergebnis später wieder an eine Speicheradresse geschrieben werden. Für die Verarbeitung müssen die Bits daher in den Prozessor transportiert werden. Diese Aufgabe übernimmt ein Datenbus, der die Bits in den Prozessor kopiert. Der Prozessor hat eigene Speicherfelder, mit denen er sehr, sehr schnell operieren kann. Dies geht aber nur so schnell, weil die Anzahl begrenzt ist. Stellen Sie sich vor, Sie haben einen Aktenschrank, bestehend aus Ordnern, in denen Daten stehen, z. B. Bilanzdaten. Dieser Schrank ist wie Ihr RAM. Sie können schnell auf einen der Ordner zugreifen und eine bestimmte Seite aufschlagen. Egal welchen Ordner Sie nehmen, die Zugriffszeit ist (ungefähr) gleich lang. Aber doch recht langsam, um überhaupt auf die Daten zuzugreifen. Wenn Sie dann den Ordner geöffnet haben, dann können Sie die gerade relevanten Daten auf ein Blatt Papier kopieren. Die Daten, die jetzt auf dem Papier stehen, können Sie selbst (und Sie sind jetzt der Prozessor, der Berechnungen ausführt) sehr viel schneller verarbeiten als die Daten aus den Ordnern. Das geht aber nur, weil auf dem Blatt Papier nur wenige Daten stehen, und Sie direkt damit arbeiten können. Und wenn Sie ein Ergebnis berechnet haben und gerade nicht mehr brauchen, dann archivieren Sie es – für die spätere Verwendung – wieder in einem der Ordner in Ihrem Aktenschrank.

Während das Speichern und Verändern auf dem Blatt Papier sehr schnell geht (ähnlich wie bei den Speicherregistern eines Prozessors), dauert der Zugriff auf die Daten im Ordner etwas länger (ähnlich wie beim Hauptspeicher des Computers). Allerdings ist auch der Hauptspeicher nicht beliebig groß. In Ihren Aktenschrank passt nur eine begrenzte Anzahl von Ordnern. Auch wenn moderne Aktenschränke immer größer werden (wir meinen natürlich: moderne Computer immer mehr RAM haben), so gibt es doch Grenzen, wenn wir sehr, sehr viele Daten speichern möchten. Aus diesem Grund gibt es neben dem Hauptspeicher noch verschiedene Arten des Massenspeichers. Dieser ist im Vergleich zum RAM relativ langsam. Beispiele für Massenspeicher sind Festplatten, USB-Sticks oder auch das Speichern in der Cloud. Um in der Analogie der Aktenordner zu bleiben: Der Massenspeicher ist das große Archiv im Keller. Dort passt viel, viel mehr hinein als in einen Aktenschrank. Aber der Zugriff auf die dortigen Ordner dauert auch nochmals etwas länger. Und wenn Sie einen Cloud-Dienst nutzen, dann verlagern Sie das Speichern der Daten (oder Akten) an einen entfernten Ort. Das kann durchaus sinnvoll sein, da dort noch viel, viel mehr Speicher zur Verfügung steht. Sie können ja auch Aktenordner in externen Speicherhäusern lagern. Dabei dauern An- und Abtransport noch einmal etwas länger. Und Sie sind auf eine gute Transportinfrastruktur angewiesen – eine schnelle Internetverbindung, die hoch verfügbar sein sollte. Und Sie müssen dem Betreiber des Speicherhauses trauen, dass die Daten vor fremden Zugriffen geschützt werden – ähnlich wie Sie einem Cloud-Anbieter trauen müssen, dass Ihre Daten nicht gehackt oder gar der Cloud-Anbieter selbst in die Daten schaut.

3.3 Heap und Stack

Der Hauptspeicher mit seinen Bits und Bytes lässt sich, wie wir bereits am Beispiel unterschiedlich großer Türen und Arrays gesehen haben, unterschiedlich organisieren. Eine weitere wesentliche Unterscheidung ist die Organisation als Heap und als Stack. Heap (englisch für „Halde“) bedeutet, dass Daten an beliebigen, bislang ungenutzten Positionen gespeichert werden können. Das bedeutet etwa: Sie richten eine neue Tür in der gewünschten Größe dort ein, wo noch Platz ist. Das ist einerseits praktisch, da Sie an beliebiger Stelle Daten speichern können. Gleichzeitig führt dies zu einer Fragmentierung des Speichers, sodass gelegentlich aufgeräumt werden muss (was Zeit kostet). Zudem müssen Sie genau organisieren, wie Sie die Tür wiederfinden, denn sie kann sich ja überall befinden. Beim Stack (englisch für „Stapel“) wird dagegen ein Speicherbereich fester Größe reserviert, und Daten können nacheinander darauf abgelegt und wieder heruntergenommen werden. Dabei kann, wie bei einem Stapel, jeweils nur auf das oberste Element zugegriffen werden. Stapel helfen Ihnen dabei, die Zugriffsreihenfolge auf bestimmte Daten effizient zu organisieren. Wir werden später sehen, dass dies beim Aufruf von Funktionen, also dem Abarbeiten wiederkehrender Anweisungsfolgen, wichtig ist, damit die Arbeit wieder an der richtigen Stelle fortgesetzt wird. Stellen Sie sich vor, dass Sie gerade auf einem Blatt Papier eine komplizierte Berechnung anstellen. An einer bestimmten Stelle benötigen Sie eine Nebenrechnung. Für diese Nebenrechnung verwenden Sie ein weiteres Blatt Papier. Dies entspricht einem neuen Speicherblock auf dem Stack (oft als Stack-Frame bezeichnet). Ihr zusätzliches Blatt Papier für die Nebenrechnung liegt wie auf einem Stapel über Ihrer eigentlichen Berechnung. Alle Zahlen und Werte, die Sie auf diesem zusätzlichen Blatt Papier aufschreiben oder verändern, haben keine Auswirkungen auf das darunter liegende Blatt Papier. Wenn Sie mit Ihrer Nebenrechnung fertig sind, entfernen Sie das zusätzliche Blatt Papier wieder und machen genau an derselben Stelle auf dem darunter liegenden Blatt Papier weiter.

3.4 Programme als Code schreiben statt als Zahlenfolgen

Wir haben also gesehen, dass der Speicher, der nur aus Nullen und Einsen besteht, auf unterschiedliche Weise organisiert wird. Zudem können die Nullen und Einsen unterschiedlich interpretiert werden. Die Zahlen können unterschiedliche Dinge repräsentieren: Datenwerte und Instruktionen für den Prozessor. Darüber hinaus können diese Zahlen auch Adressen für einen Speicherbereich darstellen. Über diese Adressen lassen sich andere Speicherbereiche referenzieren oder festlegen, wo im Speicher die nächste Instruktion für die Ausführung durch den Prozessor liegt. Dieses ganze Zahlenwirrwarr erscheint aber recht kompliziert, wenn wir doch eigentlich unsere Welt abbilden wollen, indem wir Geschäftsprozesse modellieren, Spiele schreiben, Videos oder Bilder verarbeiten, Texte schreiben oder Nachrichten verwalten und versenden möchten.

Und genau aus diesem Grunde gibt es höhere Programmiersprachen wie Kotlin, mit denen wir Programme in einer Sprache festlegen können, die sowohl für uns verständlich als auch für den Computer interpretierbar ist. Dabei abstrahieren wir von den konkreten Instruktio-

nen, die wir dem Prozessor geben müssen, auf für uns bedeutungsvolle Konzepte, wie z. B. der Ausgabe eines Textes auf dem Bildschirm. Kotlin ermöglicht z. B. die Ausgabe des Worts „Hallo“ auf dem Bildschirm mit folgender Zeile Code:

```
println("Hallo")
```

Der Prozessor kann damit aber nicht direkt etwas anfangen. Wir haben lange darüber diskutiert, dass ein Prozessor nur Instruktionen ausführen kann, die als Zahlen im Speicher liegen. Es muss also einen Weg geben, um diese Zeile Code in solche Zahlen zu übersetzen. Genau dies ist die Aufgabe von *Interpretern* und *Compilern*.

4 Interpreter und Compiler

Interpreter führen Anweisungsfolgen wie z. B. die `println`-Funktion direkt aus, indem sie bei der Abarbeitung des Programms die Bedeutung des Textes interpretieren und dann entsprechende Berechnungen durchführen. Ein Interpreter ist also selbst ein Programm, das ausgeführt werden kann. Der Interpreter liest dann ein anderes Programm ein, das als Quelltext vorliegt. Quelltext ist dabei der Programmcode, der in einer bestimmten Programmiersprache verfasst wurde, also z. B. in Kotlin. Dieser Quelltext kann in einer Datei vorliegen, die eingelesen wird, oder auch direkt in einer Entwicklungsumgebung eingetippt werden, wie dies z. B. in der Online-Umgebung von Kotlin geschieht (*https://play.kotlinlang.org*).

Der Interpreter verarbeitet dann diesen Text. Damit der Interpreter weiß, wo er mit der Arbeit beginnen kann, gibt es Einstiegspunkte, die festlegen, wo begonnen werden soll. Bei einigen Programmiersprachen ist dies die erste Zeile, die in einer Datei steht. Bei Kotlin ist dagegen der Einstiegspunkt eine Funktion, die den Namen `main()` tragen muss. Mit Funktionen werden wir uns noch sehr intensiv beschäftigen. Für den Moment können wir uns eine Funktion einfach als einen ausführbaren Codeschnipsel vorstellen. Und die `main`-Funktion ist der Codeschnipsel, der beim Programmstart ausgeführt werden soll:

```
fun main() {
    println("Hallo")
}
```

Beim Ausführen wird also mit der `main`-Funktion begonnen. In dieser Funktion können dann einzelne Anweisungen und Ausdrücke stehen, die nacheinander ausgeführt werden. Ein Interpreter stellt in diesem Beispiel fest: „Aha, ich soll erst einmal etwas auf dem Bildschirm ausgeben. Dies wird durch `println()` festgelegt. Was ausgegeben wird, ist in diesem Fall durch einen Parameter festgelegt, nämlich `"Hallo"`."

Wenn der Interpreter auf eine Anweisung stößt, die er nicht versteht, dann bricht er ab und meldet eine Fehlermeldung. Ein Interpreter arbeitet einen Quellcode also quasi *on the fly* ab, also während des Programmablaufs. Das ist im Prinzip wie ein Simultanübersetzer, der die Aussagen aus einer Sprache (Sprache des Quelltexts) live in eine andere Sprache (Sprache der Maschine) übersetzt.

Ein Compiler hat dagegen eine andere Arbeitsweise. Auch er übersetzt einen Quelltext in Code, der von der Maschine ausgeführt werden kann. Allerdings übersetzt der Compiler das Programm vollständig in Maschinencode, bevor es ausgeführt werden kann. Somit kann der Compiler auch schon Fehler melden, bevor überhaupt die Programmausführung startet.

Ein Compiler ist also kein Simultanübersetzer, sondern ein Programm, das einen Quelltext vollständig in einen Maschinencode übersetzt.

Definition *Compiler*

Ein Compiler ist ein Programm, welches ein Programm in Form von Quelltext entgegennimmt und ein anderes Programm erzeugt. Bei diesem Prozess werden eine Menge Überprüfungen und Optimierungen durchführt. Wenn das Programm keine Syntax- und Typfehler hat, kommt am Ende ein Programm heraus, welches der Computer ausführen kann.

In beiden Fällen – beim Interpreter wie beim Compiler – findet aber eine Übersetzung statt. Wenn Sie in der Experimentierumgebung von Kotlin eine Zeile Code eingeben und ausführen lassen, dann wird im Hintergrund übrigens der gesamte (wenn auch sehr kurze) Quellcode kompiliert (d. h. in Maschinencode) übersetzt. Weil dies so schnell geht und auch auf bereits eingegebene Werte zugegriffen werden kann, scheint der Ablauf wie bei einem Interpreter zu sein. Der Unterschied zwischen einem Interpreter und einem Compiler verschwindet also fast, wenn es sich um die Übersetzung sehr kurzer Quelltexte handelt. Das ist ähnlich wie der Unterschied zwischen einem Simultanübersetzer und einem Übersetzer, der einen Text vollständig übersetzt. Wenn der vollständige Text sehr kurz ist, dann gibt es (gefühlt) keinen Unterschied mehr.

4.1 Virtuelle Maschinen, Bytecode und Maschinencode

Bei Kotlin wird der Quellcode grundsätzlich kompiliert. Allerdings wird der Quellcode nicht direkt in Maschinencode übersetzt, der unmittelbar von einem Prozessor ausgeführt werden kann. Denn es gibt viele verschiedene Rechnerarchitekturen und somit auch verschiedene Prozessoren, die zum Einsatz kommen. Und während es bei Buchstaben standardisierte Codes gibt, die festlegen, welcher Buchstabe durch welche Zahl repräsentiert wird, gibt es bei den Prozessoren verschiedener Hersteller unterschiedliche Formen, wie Instruktionen als Zahlen codiert werden. Das liegt unter anderem daran, dass die Prozessoren unterschiedliche Befehlssätze haben, um z. B. komplexere Operationen in einem Schritt auszuführen. Es gibt hier konkurrierende Systeme, die sich durch eine unterschiedliche bzw. unterschiedlich schnelle Verarbeitung der Daten an die Spitze setzen möchten.

Um von diesen Unterschieden verschiedener Prozessortypen zu abstrahieren, kommen oft *virtuelle Maschinen* zum Einsatz. Eine virtuelle Maschine besteht ähnlich wie eine reale Computer-Maschine vor allem aus einem Prozessor zum Verarbeiten von Instruktionen und einer Speicherverwaltung. Bei einer virtuellen Maschine ist dieser Prozessor jedoch keine Hardware-Komponente, sondern selbst ein Stück Software, also ein Programm. Dieser virtuelle Prozessor löst das Problem, dass unterschiedliche Hardware-Prozessoren mit unterschiedlichen Befehlssätzen ausgestattet sind. Die virtuelle Maschine stellt nämlich einen einheitlichen Befehlssatz zur Verfügung. Die virtuelle Maschine kann dann für verschiedene Hardware-Systeme umgesetzt werden. Damit wird erreicht, dass auf verschiedenen

Hardware-Plattformen derselbe Code ausgeführt wird. Dieser Code ist im Prinzip ein Zwischencode, der als Bytecode bezeichnet wird. Für den Compiler bedeutet dies, dass nicht zwischen verschiedenen Hardware-Plattformen unterschieden werden muss, sondern für eine virtuelle Maschine übersetzt werden kann. Zudem übernimmt die virtuelle Maschine die Speicherverwaltung und sorgt dafür, dass keine Speicherbereiche (im realen Hardware-RAM) belegt werden, die außerhalb der virtuellen Maschine liegen. Eine sehr weit verbreitete virtuelle Maschine ist die *Java Virtual Machine* (JVM). Diese Architektur wurde ursprünglich für die Programmiersprache Java entwickelt. Es gibt verschiedene Umsetzungen von dieser virtuellen Maschine – einerseits von verschiedenen Herstellern, andererseits für verschiedene Hardware-Plattformen. So gibt es eine JVM für Mac-Systeme, für Windows-Systeme, für Unix-Systeme, aber auch für Android-Smartphones. Die JVM ist nichts anderes als eine *Laufzeitumgebung*, die Programmcode ausführen kann. Dieser Programmcode verwendet den Befehlssatz der JVM und wird als Bytecode bezeichnet.

Ein Compiler kann nun den Quelltext einer von Menschen gut lesbaren Sprache (wie Kotlin oder Java) in Bytecode für die JVM übersetzen. Die JVM interpretiert oder übersetzt diesen Bytecode dann in spezifische Instruktionen für die jeweilige Hardware-Plattform – egal welcher Prozessor dort zum Einsatz kommt. Damit ist es theoretisch möglich, denselben Quellcode nur ein einziges Mal zu übersetzen und dann über die JVM auf verschiedenen Hardware-Plattformen laufen zu lassen. Allerdings ist es so, dass einzelne Bibliotheken – also zur Verfügung gestellte fertige Funktionalitäten – nur für bestimmte Plattformen verfügbar sind. Das liegt daran, dass z. B. einerseits sowohl für Mac und Windows jeweils grafische Benutzeroberflächen mit Fenstern als auch die Ein- und Ausgabe über eine Konsole zur Verfügung stehen. Hier gibt es auch gemeinsame Bibliotheken (z. B. Java FX), mit denen kompilierte Programme entwickelt werden können, die 1:1 auf beiden Plattformen laufen. Andererseits gibt es aber etwa für Android-Systeme spezifische Bibliotheken, die eine spezifische Gestaltung der Benutzeroberfläche berücksichtigen und auch auf für Smartphones spezifische Funktionalitäten Zugriff gewähren, etwa die Bewegungssensoren oder das Starten eines Telefonanrufs.

4.2 Kotlin – eine Sprache, viele Plattformen

Kotlin ist als moderne Programmiersprache so ausgelegt, dass sie auf möglichst vielen Plattformen läuft. Es gibt daher verschiedene Compiler für die unterschiedlichen Zielplattformen. Mit anderen Worten: Der in Kotlin geschriebene Quelltext kann sowohl für die Java Virtuelle Maschine als auch für andere Systeme übersetzt werden. Insbesondere ist eine Übersetzung in JavaScript-Code sowie auch das Kompilieren für native Hardware-Plattformen möglich.

Der häufigste Fall ist derzeit noch die Kompilierung von Kotlin-Programmen für die Java Virtuelle Maschine. Darauf werden wir uns auch in diesem Buch konzentrieren. Dabei ist es keine Einschränkung, dass die JVM ursprünglich einmal für die Programmiersprache Java entwickelt wurde. Inzwischen gibt es längst mehrere andere Sprachen, die auch für die JVM übersetzt werden können. Es ist sogar ein Vorteil, dass Kotlin für die JVM übersetzt werden kann. Zum einen übernehmen wir damit die Plattformunabhängigkeit (mit den oben ge-

nannten Einschränkungen), die auch für Java galt. Zum anderen klappt das Zusammenspiel mit existierenden Java-Programmen (die Interoperabilität der Programmiersprachen Kotlin und Java) reibungslos. Man kann Kotlin vollständig nutzen, ohne Java zu kennen. Wenn Sie jedoch Java-Kenntnisse oder bereits Code in Java entwickelt haben, dann können Sie dieses Wissen bzw. diesen Code problemlos weiter verwenden.

Für häufig wiederkehrende Aufgaben gibt es zudem bereits Standardbibliotheken, die umfangreiche Funktionalität bereitstellen. Wenn Sie z. B. einen Algorithmus benötigen, um Daten zu sortieren, oder ein Bild in einem bestimmten Format laden oder speichern möchten, dann macht es wenig Sinn, immer und immer wieder denselben Algorithmus zu programmieren. Stattdessen gibt es Bibliotheken, die eingebunden werden können. Gerade für Java gibt es zahlreiche Bibliotheken, die sehr spezielle Funktionalitäten bereitstellen, z. B. die Verbindung zu einem bestimmten Server oder die Nutzung bestimmter Online-Dienste. Alle Bibliotheken, die für Java verfügbar sind, können Sie auch in Kotlin nutzen, wenn Sie für die JVM kompilieren.

Zudem ist es möglich, Kotlin-Programme in die Programmiersprache JavaScript zu übersetzen. JavaScript ist eine Programmiersprache, die bei der Ausführung interpretiert wird. Wenn also aus dem Kotlin-Quelltext ein JavaScript-Programm entsteht, dann kann dieses sofort in einer JavaScript-Umgebung ausgeführt werden. Die bekannteste Umgebung zur Ausführung von JavaScript-Programmen ist Ihr Web-Browser. Viele Interaktionen auf einer Webseite, die im Browser ausgeführt werden, sind als ausführbare JavaScript-Programme definiert. Sie können Kotlin also auch nutzen, um interaktive Webanwendungen zu entwickeln. JavaScript läuft aber nicht nur im Web-Browser (dem Client), sondern immer häufiger auch auf Servern. Insbesondere die Node.js-Plattform ist eine Serverarchitektur, bei der die Backend-Entwicklung mit JavaScript geschieht. Mit Backend ist der Teil einer Webanwendung gemeint, die nicht in Ihrem Web-Browser abläuft, sondern auf dem Webserver des Anbieters. Wenn Sie z. B. bei einem Online-Shop etwas bestellen, dann läuft ein Teil des Systems auf Ihrem Rechner ab (z. B. blendet das Klicken auf ein Produkt zusätzliche Informationen ein oder legt es in den Einkaufswagen), und der andere Teil läuft auf dem Server des Anbieters (z. B. Informationen über ein Produkt bereitstellen, Abwickeln der Bestellung).

Mit Kotlin geschriebene Programme lassen sich aber auch direkt in den Maschinencode einer Plattform übersetzen. Diese Option wird als *Kotlin Native* bezeichnet, weil nativer, auf einer bestimmten Maschine ausführbarer Code entsteht. Dies funktioniert inzwischen auch schon für die iOS-Plattform sehr gut. Allerdings befinden sich einige Merkmale von Kotlin Native noch in der Entwicklung und werden erst nach und nach bereitgestellt. Dennoch lässt sich schon jetzt sagen, dass Kotlin auch hier zukunftsfähig ist.

Wir werden uns in diesem Buch auf die Kompilierung von Kotlin-Quellcode für die JVM konzentrieren – sowohl für Anwendungen auf einem regulären Computer (Windows-PC, Mac oder Linux) als auch für Android-Apps. Trotzdem ist es wichtig zu wissen, dass Kotlin sich für verschiedene Plattformen übersetzen lässt.

5 Syntax, Semantik und Pragmatik

Damit die Übersetzung klappt, muss natürlich auch der Quellcode in einer für den Übersetzer verständlichen Sprache vorliegen. Stellen Sie sich vor, Sie haben jemanden, der Deutsch und Türkisch spricht. Diese Person kann deutsche Texte ins Türkische übersetzen und umgekehrt. Wenn Sie dagegen einen Text haben, der im Spanischen vorliegt, wird Ihnen die Person vermutlich nicht weiterhelfen können.

Genauso ist es mit Compilern. Ein Kotlin-Compiler versteht nur syntaktisch korrekten Quellcode. Nur dieser Code kann übersetzt werden. Die Syntax ist dabei ein **Regelsystem von gültigen Zeichen**. Eine ordentliche Syntax unserer natürlichen Sprache ist z. B.:

```
Am Montag scheint die Sonne.
```

Keine gültige Syntax für einen ordentlichen Satz sind dagegen die folgenden Beispiele:

```
!!! Sonne, Am scheinen Montag. Das
Montags, Sonnen, scheinen
A. M. s. d. S.
JISOJSK!!!
```

Diese Sätze verletzen unsere Grammatik und verwenden teils keine eindeutigen bzw. nicht existierende Begriffe. Sie sind **syntaktisch falsch**. Während wir bei einigen Varianten noch erraten können, was gemeint ist, verlangt ein Compiler stets eine gültige Syntax. Compiler raten nicht.

5.1 Syntax

Compiler benötigen wohlgeformte Syntax. Diese bestehen beispielsweise aus

- fest definierten Schlüsselwörtern, dem Vokabular der Sprache Kotlin, wie z. B. `if`, `class` oder `fun`,
- selbst eingeführten Bezeichnern wie z. B. Namen für Datenwerte oder Funktionen,
- Operationen wie z. B. `+`, `-`, `*`, `/` und
- feststehenden Datenwerten wie z. B. `42` oder `"Hallo"`.

Definition *Syntax*

Die Syntax ist ein Regelsystem von gültigen Zeichen für eine Sprache.

Wir haben bereits ein Beispiel für eine gültige Syntax gesehen:

```
println("Hallo")
```

Dabei handelt es sich syntaktisch um einen wohlgeformten Satz im Quellcode der Programmiersprache Kotlin. Es wird der Bezeichner `println` für eine Kotlin-Funktion zur Ausgabe auf der Konsole verwendet. Was ausgegeben wird, kann über einen Parameter festgelegt werden. Parameter werden in runden Klammern hinter die Funktion geschrieben. Feststehende Zeichenketten können in Kotlin mit doppelten Anführungszeichen beginnen und enden.

Beispiele für eine ungültige Syntax sind:

```
p r i n t l n ("Hallo") // Leerzeichen zwischen Buchstaben des Bezeichners nicht
     erlaubt
"Hallo" () println // falscher Aufbau, ähnlich wie bei der falschen Grammatik oben
```

Die Syntax zum Deklarieren und Zuweisen einer Variable ist:

```
val hallo = "Hallo"
println(hallo)
```

Dabei folgt die Definition der Variablen `hallo` hier auch einer festgelegten Syntax. Folgendes wäre nicht möglich:

```
Setze den Variablenwert hallo auf den Wert "Hallo"
```

Das ist zwar für uns verständlich, aber es entspricht nicht der Kotlin-Syntax. Und somit kann der Compiler diesen Satz – im Gegensatz zu uns – nicht verstehen und übersetzen. Was es genau mit Variablen auf sich hat, schauen wir uns noch später an.

5.2 Semantik

Ein syntaktisch korrektes Programm muss noch lange nicht richtig sein. Schauen wir uns als Beispiel einmal folgenden grammatikalisch richtigen Satz an:

```
Am Montag regnet die Sonne.
```

Dieser Satz ist grammatikalisch in Ordnung und entspricht der Syntax der deutschen Sprache. Aber er ist natürlich Quatsch! Die Sonne kann ja nicht regnen. Selbst wenn es regnet und stürmt, dann ist es nicht die Sonne, die regnet. Ein anderes Beispiel für einen grammatikalisch richtigen, aber sinnfreien Ausdruck ist der Spruch:

```
Nachts ist es kälter als draußen.
```

Dass es die natürliche Sprache erlaubt, solche syntaktisch richtigen, aber inhaltlich sinnfreien Aussagen zu treffen, ist die Grundlage vieler Wortwitze. Bei der Softwareentwicklung sind solche Witze jedoch weniger angebracht. Der Compiler lacht nicht, sondern übersetzt den Code, der stumpf ausgeführt wird. Nehmen wir folgendes einfaches Beispiel:

```
val a = 10
val b = 20
val multiplication = a + b
println("Das Ergebnis von a * b ist $multiplication")
```

Hier wird in der letzten Zeile der Inhalt der Variablen `multiplication` ausgegeben. Es wird behauptet, dass es sich um das Ergebnis der Berechnung von `a * b` handelt. Allerdings weisen wir der Variable `multiplication` nicht etwa `a * b`, sondern `a + b` zu.

Syntaktisch ist dies gar kein Problem. Wir können auf der rechten Seite des Gleichheitszeichens (=) eine beliebige Berechnung anstellen und das Ergebnis einer Variablen zuweisen, die wir später wiederverwenden. Aber die Bedeutung ist hier natürlich nicht richtig. Die Bedeutung eines Programms nennt man *Semantik*.

- `a + b` hat die Semantik, dass die Werte von a und b addiert werden sollen.
- `a * b` hat die Semantik, dass die Werte von a und b multipliziert werden.

Beide Operationen sind syntaktisch korrekt. Das hier gezeigte Beispiel ist in der Tat recht trottelig. Dennoch sind die meisten Programmfehler semantischer Art. Wenn wir z. B. den Prozentwert berechnen wollen und dabei vergessen, durch 100 zu dividieren (Prozent heißt ja gerade pro Hundert), dann ist dies kein Syntaxfehler, sondern ein Semantikfehler. Wenn wir vergessen, eine Einzahlung auf einem Konto durchzuführen, dann ist dies kein Syntaxfehler (es gibt ja gar keine Anweisung für die Einzahlung), sondern ein Semantikfehler. Falsche Berechnungen, falsche oder vergessene Aktionen, all dies sind Semantikfehler.

Definition *Semantik*

Die Semantik ist die Bedeutung einer Sprache.

Während der Compiler bereits bei der Übersetzung Alarm schlägt, wenn etwas syntaktisch nicht in Ordnung ist, kann der Compiler keine semantischen Fehler finden. Denn der Compiler kann ja nicht wissen, was Sie wirklich wollen. Der Compiler ist ja nur ein dummes (wenn auch sehr leistungsfähiges) Programm, das Ihre Quelltexte in Maschinen- oder Bytecode übersetzt. Er hinterfragt nicht die Sinnhaftigkeit.

Semantikfehler zeigen sich in der Regel erst zur *Laufzeit*. Mit Laufzeit ist die eigentliche Ausführung Ihres Programms gemeint. Also der große Moment, in dem ein Prozessor bestimmte Speicherzellen als Instruktionen interpretiert und die dort eingeforderten Operationen stoisch ausführt. Stoisch deswegen, weil auch die Laufzeitumgebung keine Beurteilung durchführt, ob ihr Programm gerade Sinn macht oder nicht. Allerdings gibt es ein paar Fehler, auf die eine Laufzeitumgebung uns aufmerksam machen wird, z. B. wenn wir auf Speicherbereiche oder Elemente zugreifen wollen, die es gar nicht gibt. Da solche Zugriffe oft erst zur Laufzeit berechnet werden, merkt der Compiler nichts von diesem Fehler. Und die Laufzeitumgebung stellt den Fehler erst dann fest, wenn er auftritt – also wenn eine Instruktion ausgeführt werden soll, die unerlaubt ist. Ein weiteres typisches Beispiel hierfür ist die Division durch 0. Diese Operation ist mathematisch nicht erlaubt. Ob jedoch der Nenner bei einer Berechnung tatsächlich 0 ist, zeigt sich erst zur Laufzeit.

Damit möglichst viele Fehler, die erst zur Laufzeit entdeckt werden können, bereits erkannt werden, bevor eine Software ausgeliefert wird, kann man Testfunktionen und Testfälle schreiben. Diese simulieren den regulären Programmablauf zur Laufzeit und testen für

verschiedene Werte (insbesondere für Grenzwerte und kritische Werte), ob das Programm auch zur Laufzeit das Richtige berechnet. Für das obige Beispiel könnten wir etwa testen, ob in `multiplication` der erwartete Wert `200` steht. Dies wäre das Ergebnis, wenn die beiden Variablen `a` und `b` semantisch korrekt miteinander multipliziert werden.

5.3 Pragmatik

Neben der semantischen Ebene gibt es noch die *pragmatische Ebene*. Diese zieht zur Beurteilung der Bedeutung auch noch den Kontext hinzu. Nehmen wir folgendes Beispiel:

```
val a = 8000
val b = a + 4000
```

In `b` muss dann der Wert `12000` stehen. Dies wäre dann ein syntaktisch und semantisch korrektes Programm. Wenn wir `b` nun aber als die Anzahl der Personen, die sich in einer Veranstaltungshalle befinden, interpretieren, dann ist `b` nur dann korrekt, wenn auch mind. 12000 Personen dort hinein passen. Die pragmatische Korrektheit eines Programms ist am schwierigsten zu überprüfen. Sie lässt sich nicht immer durch Testfälle überprüfen, sondern erfordert oft die Erprobung im realen Einsatz.

- **Syntaktisch** kann der Compiler feststellen, dass `a + 4000` ein korrekter Ausdruck ist.
- **Semantisch** lässt sich prüfen, ob `b` den richtigen Wert `12000` speichert.
- **Pragmatisch** ist `b` nur dann richtig, wenn `12.000` ein gültiger Wert ist, also hier `12.000` Personen in die Halle passen.

In diesem Beispiel lässt sich das Maximum der Veranstaltungsbesucher noch als weiterer Wert festlegen. Es gibt jedoch auch viele Beispiele, bei denen die Gültigkeit eines Wertes auf das Alltagswissen der Anwender ankommt. Nehmen wir an, dass tatsächlich 16.000 Personen in die Halle passen. Was bedeutet dies dann? Ist die Halle dann bei 12.000 Personen schon gut oder nur mäßig besucht?

Den Unterschied zwischen Syntax, Semantik und Pragmatik kann man noch einmal an diesem Satz verdeutlichen: Der Baum ist um 10 cm gewachsen.

- **Syntax**: Der Satz ist in unserer natürlichen Sprache syntaktisch vollkommen in Ordnung.
- **Semantik**: Auch semantisch ist er klar besetzt: Der Baum ist jetzt größer als vorher (und zwar 10 cm größer).
- **Pragmatik**: Aber pragmatisch kann er ganz unterschiedliche Dinge bedeuten. Herr Müller freut sich, dass der Baum wächst und noch mehr Blüten trägt. Herr Meier ärgert sich, weil der Baum mehr Schatten wirft und noch mehr Blätter weggefegt werden müssen.

Während der Laufzeit eines Programms spielt die Pragmatik ebenfalls eine große Rolle. Denn sie beeinflusst die Interaktionen mit den Benutzern. Welche Bedeutung hat es z. B., wenn der Benutzer auf der Konsole eine bestimmte Anweisung eintippt? Welche Bedeutung hat es, wenn der Benutzer irgendwo hin klickt? Welche Aktionen möchte er ausführen?

Um solche pragmatischen Entscheidungen oder Erläuterungen direkt im Quellcode unterzubringen, bieten alle Programmiersprachen die Möglichkeit, Kommentare in den Code zu

schreiben. Ein Kommentar wird bei der Übersetzung ignoriert. Er dient nur den Entwicklerinnen und Entwicklern als Erläuterung.

In Kotlin werden einzeilige Kommentare mit zwei Schrägstrichen `//` eingeleitet. Mehrzeilige Kommentare werden mit `/*` begonnen und mit `*/` beendet:

```
// Einzeiliger Kommentar wird oft zur Erläuterung der
// nächsten Zeile verwendet. Beispiel:

// In Deutschland ist man ab 18 volljährig
val minage = 18

/* Kommentare können auch über mehrere Zeilen gehen.
   Das bietet sich für die ausführliche Dokumentation von Funktionen
   oder die Erläuterungen von Algorithmen, Spezifikationen, Entscheidungen usw. an.
   Sie sollten Kommentare jedoch nicht dazu verwenden, offensichtliche Dinge
   zu erläutern. Guter Programmcode, insbesondere in Kotlin, dokumentiert sich
   zu großen Teilen selbst, indem sinnvolle Namen für Funktionen und Variablen
   verwendet werden. */
```

6 Eingabe – Verarbeitung – Ausgabe

Zur Laufzeit ist es übrigens so, dass nicht nur die existierenden Daten im Hauptspeicher durch entsprechende Instruktionen verändert werden. Es gibt auch ein Zusammenspiel von Ein- und Ausgabedaten. Dabei ist der klassische Ablauf folgender: Es werden Daten eingegeben, diese werden verarbeitet und schließlich ausgegeben.

Für Programme, die auf der Konsole ausgeführt werden, bedeutet dies: Es wird etwas über die Tastatur eingegeben. Diese Eingabedaten werden in Speicherzellen geschrieben. Der Prozessor kann sie dann verarbeiten und ein Ergebnis berechnen. Das Ergebnis wird dann wiederum in Speicherzellen geschrieben. Bei der Ausgabe werden die Daten aus der Speicherzelle gelesen und auf der Konsole ausgegeben.

Ein in Kotlin geschriebenes Programm kann nun übersetzt werden und Ausgaben auf einer Konsole anstoßen. Das für jede Programmiersprache obligatorische erste Programm gibt z. B. `"Hallo Welt"` auf der Konsole aus:

Bei grafischen Benutzeroberflächen geschieht im Prinzip dasselbe. Allerdings sind die Eingaben hier nicht Buchstaben auf der Tastatur, sondern Mausbewegungen, Mausklicks usw. Jede Mausbewegung führt zu einer Veränderung der gespeicherten Daten – des internen Zustands Ihres Programms. Wenn an einer bestimmten Stelle geklickt wird, dann kann bei einem bestimmten Zustand (z. B. Mauscursor befindet sich über einem Button) von der Laufzeitumgebung ein Ereignis ausgelöst werden. Dieses Ereignis „Button wurde angeklickt“ ist nichts anderes als eine weitere Eingabe, die verarbeitet werden kann. Die Ausgabe erfolgt

dann meist durch Manipulation der grafischen Oberfläche, z. B. Ändern von Beschriftungen, Wechsel zu einem anderen Screen oder Starten einer Animation. Das Programm wird allerdings nicht nach der Ausgabe beendet (wie dies bei Konsolenprogrammen meist der Fall ist), sondern wartet direkt auf die nächste Eingabe (z. B. nächste Mausbewegung). Dies bezeichnet man als Event-Loop. In diesem Event-Loop wird wieder und wieder geschaut, ob sich bestimmte Speicherbereiche geändert haben (etwa der Speicherbereich, der angibt, wo sich der Mauszeiger befindet). Sobald sich eine interessante Veränderung ergibt, kann eine Verarbeitung angestoßen werden. Da es sehr viele Veränderungsmöglichkeiten gibt (Mausbewegungen, Mausklicks an bestimmten Stellen), definiert man verschiedene Verarbeitungsstrategien, um auf solche Events zu reagieren. Aber auch hier gilt, was anfangs gesagt wurde: Es handelt sich stets nur um systematische Berechnungen von Zahlen.

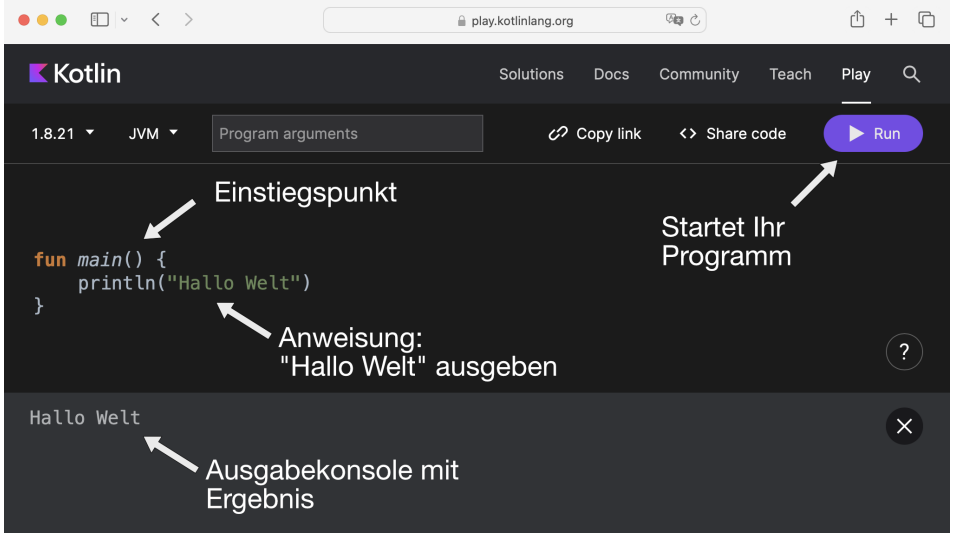
Diese Berechnungen dienen dem Erfüllen einer Aufgabe bzw. dem Lösen eines Problems. Man bezeichnet solche formal definierten Lösungswege als Algorithmen. Ein Algorithmus legt für Computer also fest, wie neue Daten berechnet werden sollen. Ein Programm besteht nun aus vielen Algorithmen, die zusammenspielen.

7 Los geht's

Wie wir gesehen haben, müssen wir uns nicht mit lauter Nullen und Einsen auseinandersetzen. Stattdessen helfen uns Compiler dabei, einen Quelltext zu übersetzen. Der Quelltext kann sich in einer oder mehreren Dateien auf Ihrem Rechner befinden. Der Compiler übersetzt diese Dateien in Byte- oder Maschinencode.

Das Schreiben der Quelltexte kann in Prinzip mit jedem Editor geschehen. Ja, selbst der einfache Texteditor von Windows ist dafür geeignet, ein Kotlin-Programm zu schreiben, denn es handelt sich erst einmal um reine Textdateien. Diese Textdateien können wir dann über die Kommandozeile kompilieren. Jetzt benötigen wir für die Ausführung noch eine JVM. Diese können wir ebenfalls über die Kommandozeile starten und dabei angeben, welches kompilierte Programm ausgeführt werden soll. Das klingt kompliziert und nicht besonders komfortabel.

Zum Glück gibt es zwei Lösungen. Zum einen können Sie eine Online-Umgebung nutzen, um die ersten kleinen Programme zu schreiben. Darin können Sie sofort loslegen, ohne irgendetwas auf Ihrem Rechner zu installieren. Für größere Programme werden wir dann eine sogenannte *integrierte Entwicklungsumgebung* verwenden, die zahlreiche weitere Vorteile bietet. Doch jetzt legen wir erst einmal mit der Online-Umgebung los:



Wenn Sie auf *https://play.kotlinlang.org* gehen, dann können Sie direkt im Web-Browser Ihre ersten Kotlin-Programme schreiben.

Jedes Programm startet in einer `main`-Funktion. Mehr zu Funktionen erfahren Sie später. Wichtig ist erst einmal, dass es immer diese `main`-Funktion gibt und man darin Programmcode schreiben kann. Machen wir das doch direkt und geben etwas auf der Konsole aus:

```
println("Hallo Welt")
```

Dieser Programmcode wird beim Programmablauf ausgeführt. Wenn Sie auf den Run-Button klicken, dann wird das Programm kompiliert, also in ausführbaren Code übersetzt, und direkt ausgeführt. Das funktioniert allerdings nur, wenn das Programm fehlerfrei ist.

Die Online-Umgebung ist eine gute Spielwiese zum Ausprobieren. Für die effektive Entwicklung benötigen Sie jedoch eine integrierte Entwicklungsumgebung, die Sie bei der Softwareentwicklung unterstützt.

7.1 Integrierte Entwicklungsumgebung

Eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, kurz IDE) unterstützt Sie bei der Entwicklung von Programmen. Sie bietet einen sehr leistungsfähigen Editor zum Schreiben des Programmcodes und verwaltet Ihre Programmstruktur. Zudem unterstützt Sie die IDE bei der Fehlersuche. Syntaxfehler werden sofort angezeigt. Des Weiteren können Sie Ihren Programmcode mit einem Debugger schrittweise durchlaufen und so Fehler suchen. Damit werden wir uns später ausführlicher beschäftigen. Die IDE hilft uns auch dabei, die richtigen Operationen und Funktionen für bereits existierende Typen zu finden, denn der Editor bietet eine Codevervollständigung. Das bedeutet: Der Editor zeigt an, welche Operationen für ein bestimmtes Objekt möglich sind. Auch unsere eigenen Variablennamen müssen wir nicht vollständig von Hand eintippen. Stattdessen reichen die Anfangsbuchstaben, und der Editor schlägt vor, welcher Name verwendet werden soll.

Betrachten wir zunächst noch einmal den Quellcode. Dieser könnte, wie gesagt, auch in einem einfachen Texteditor entwickelt werden. Doch dieser bietet uns keinerlei Unterstützung beim Schreiben. So weist der einfache Texteditor nicht darauf hin, wenn wir uns vertippt haben und syntaktisch falsche Anweisungen schreiben. Von modernen Textverarbeitungsprogrammen sind wir inzwischen Besseres gewöhnt. Diese heben Rechtschreib- und Grammatikfehler hervor und korrigieren diese teils noch beim Tippen. Allerdings helfen uns die Textverarbeitungen eben nur beim Schreiben von Texten in natürlicher Sprache.

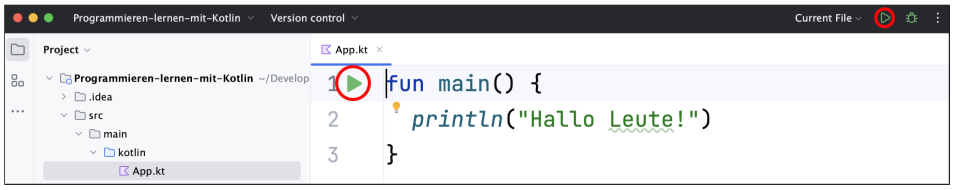
Eine integrierte Entwicklungsumgebung spricht dagegen die Sprache des Programmierens. Sie unterstützt uns dabei, syntaktisch richtige Programme zu schreiben. Dies geschieht auf unterschiedliche Weise:

- Hervorheben von syntaktischen Fehlern
- Vorschläge zum Ausbessern der Fehler
- Syntax-Highlighting zum Hervorheben verschiedener Programmelemente
- Vorschläge für die Verwendung von bereits existierenden Variablen oder Funktionen
- Codevervollständigung für bereits existierende Namen und typische Konstrukte

Die IDE analysiert schon beim Programmieren, welche Struktur unser Code hat. Dadurch kann die IDE Vorschläge unterbreiten, welche bereits existierenden Elemente wir wiederverwenden können. Das ist sehr praktisch, da uns so stets eingeblendet wird, welche Funktionen uns zur Verfügung stehen. Denn einen Großteil des Codes, den wir einsetzen, werden wir nicht selbst schreiben. Stattdessen setzen wir Standardbibliotheken ein, die effiziente und getestete Funktionen bereitstellen. Die IDE hilft uns dabei zu entscheiden, welche Funktionen wir einsetzen können und wie diese aufgerufen werden. Wenn Sie große Projekte starten, dann hilft Ihnen eine IDE auch bei der Umstrukturierung des Programmcodes. Es gibt noch zahlreiche weitere Annehmlichkeiten, die Sie während der Entwicklung Schritt für Schritt schätzen lernen.

Wir werden eine IDE namens *IntelliJ* für einen Großteil des Buches verwenden. In Teil VII, „Android“, verwenden wir *Android Studio*. Dabei handelt es sich um eine spezielle, vorkonfigurierte Variante von IntelliJ, die für die Android-Plattform optimiert wurde.

Eine Aufgabe der IDE ist es zudem, alle erforderlichen Programmteile zu übersetzen und mit den eingesetzten Bibliotheken zu verknüpfen. Wenn Sie Ihr geschriebenes Programm laufen lassen möchten, dann müssen Sie nur noch den Play-Button drücken. Der Play-Button am linken Rand des Quellcodes erscheint neben allen `main`-Funktionen, die Sie in Ihrem Projekt haben. Klicken Sie auf diesen Button, um das Projekt zu kompilieren und diese `main`-Funktion als Einstiegspunkt zu verwenden. Wenn Sie dagegen auf den Play-Button oben in der Toolbar klicken, dann wird stets die zuletzt gewählte `main`-Funktion wieder als Einstiegspunkt verwendet. Daher müssen Sie auch beim Ausführen eines Programms in einem neu angelegten Projekt auf den Play-Button links neben der `main`-Funktion klicken. So legen Sie gleichzeitig fest, welche `main`-Funktion ausgeführt werden soll.



Wenn das Quellprogramm fehlerfrei ist, dann wird es in einem Rutsch kompiliert, alles Nötige eingebunden und dann auch gleich ausgeführt. Wenn beim Übersetzen Fehler auftreten, können Sie mit einem Klick an die richtige Stelle springen. Die IDE stellt also eine enge Verknüpfung zwischen dynamischer Programmausführung und statischem Quelltext her. Dies kann sogar noch einen Schritt weiter gehen, wenn Sie mit dem Debugger arbeiten.

Syntaktische Fehler können bereits vor dem Kompilieren von der IDE angezeigt werden. So weist Sie die IDE darauf hin, wenn Sie Variablen und Daten verwenden, die es gar nicht gibt. Genauso werden Sie darauf hingewiesen, wenn Sie Operationen ausführen wollen, die es nicht gibt oder die für bestimmte Werte nicht erlaubt sind. Das sind sogenannte Typfehler.

Doch wir haben auch gesehen, dass es eine Reihe von Fehlern gibt, die sich erst zur Laufzeit zeigen. Um diese Fehler zu beseitigen, ist es meist notwendig, genau nachzuvollziehen, was eigentlich beim Ablauf des Programms geschieht. Werden die Variablen so gesetzt und verändert, wie wir dies im Kopf geplant haben? Oder haben wir etwas irgendwo vergessen?

Dies lässt sich gut nachvollziehen, wenn man den Ablauf wirklich durchspielt. Und zwar nicht nur im Kopf, sondern tatsächlich durch Ausführen des Programms. Genau dafür stellen

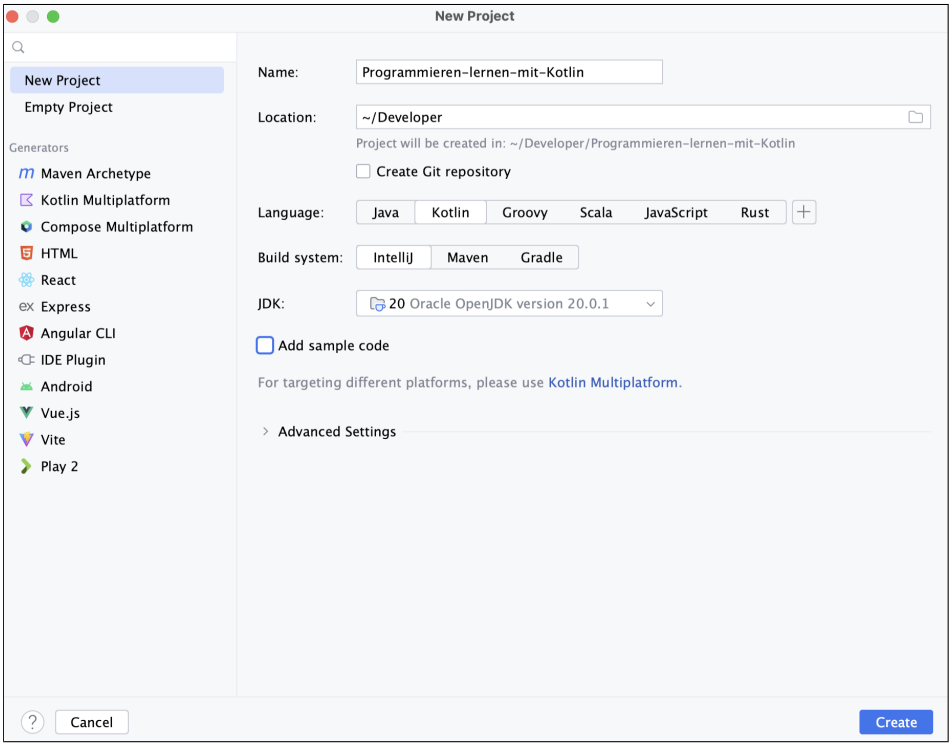
IDEs einen *Debugger* zur Verfügung. Wie Sie mit dem Debugger arbeiten, werden Sie in Kapitel 23, „Debugger“, erfahren.

Gerade für umfangreichere Programme ist das Debuggen unerlässlich, um einen Fehler aufzuspüren. Daher wollten wir es bereits an dieser Stelle kurz erwähnen. Wir haben häufig beobachtet, dass gerade Programmiereinsteiger oft versuchen, das Problem eines fehlerhaften Programms rein theoretisch und durch Nachdenken zu lösen. Ein Fehler lässt sich aber viel, viel einfacher lösen, wenn man geradezu sieht, was falsch läuft. Genau hierfür ist der Debugger da. Und er hilft Ihnen auch dabei, Programmieren zu lernen.

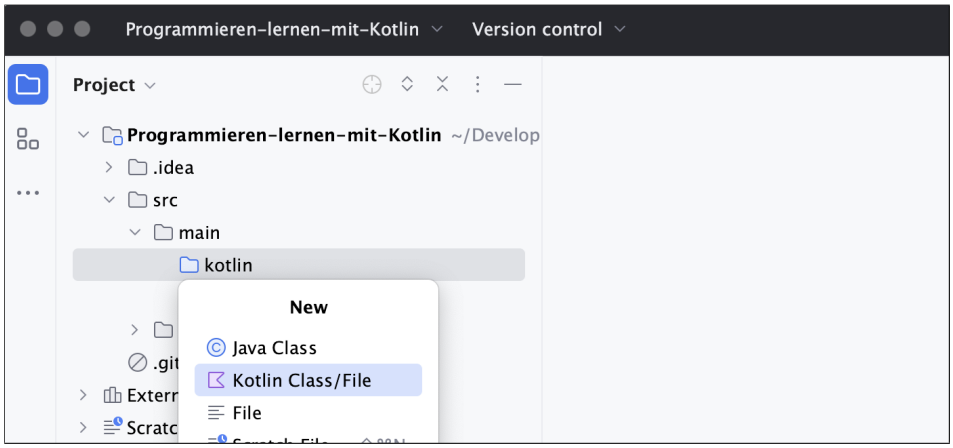
7.2 Projekt anlegen

Die IntelliJ Community Edition ist kostenlos, und Sie sollten diese auf Ihrem Rechner installieren, um darin alle Beispiele in diesem Buch als Projekt anzulegen.

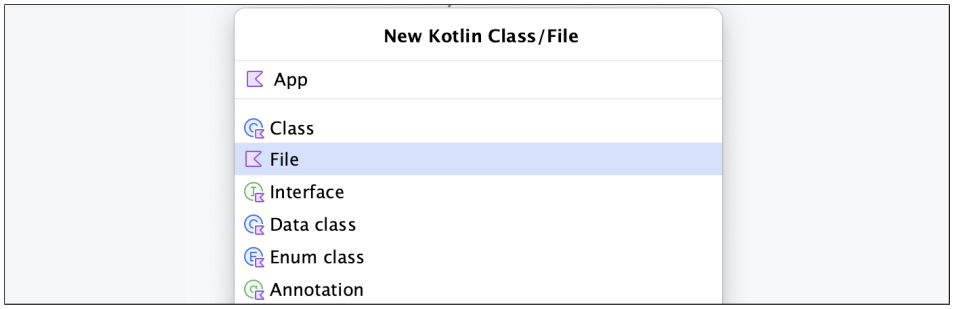
So legen Sie ein neues Projekt an. Gehen Sie im Menü auf „File -> New -> Project“. Vergeben Sie zuerst einen Namen für das Projekt und wählen Sie einen Speicherort. Setzen Sie danach die folgenden Einstellungen: „Kotlin“ als Programmiersprache, „IntelliJ“ als Build-System und das jeweils neuste „JDK“ (in diesem Fall 20). Entfernen Sie zudem den Haken bei „Add sample code“. Bestätigen Sie anschließend mit „Create“:



In dem neuen Projekt gibt es links einen Projekt-Browser. Dort finden Sie den „src“-Ordner. Darin werden alle Quelldateien (Source Code) angelegt. Klicken Sie den „src“-Order so weit durch, bis Sie beim Ordner „kotlin“ ankommen. Hier erzeugen wir nun eine neue Kotlin-Datei, indem wir mit der rechten Maustaste auf den „kotlin“-Ordner klicken und dann „New -> Kotlin Class/File“ auswählen:



Wählen Sie in der Auswahl „File“ aus und vergeben Sie einen beliebigen Namen, wie z. B. App:



Jetzt können Sie Ihr erstes eigenes Programm schreiben: