ATOMIC KOTLIN

Bruce Eckel

Svetlana Isakova

Deutsche Ausgabe

Atomic Kotlin (Deutsche Ausgabe)

Bruce Eckel und Svetlana Isakova

Dieses Buch wird verkauft unter http://leanpub.com/AtomicKotlin-de

Diese Version wurde veröffentlicht am 2024-09-18

ISBN 978-0-9818725-4-4

Dies ist ein Leanpub-Buch. Leanpub bietet Autoren und Verlagen, mit Hilfe von Lean-Publishing, neue Möglichkeiten des Publizierens. Lean Publishing bedeutet die wiederholte Veröffentlichung neuer Beta-Versionen eines eBooks unter der Zuhilfenahme schlanker Werkzeuge. Das Feedback der Erstleser hilft dem Autor bei der Finalisierung und der anschließenden Vermarktung des Buches. Lean Publishing unterstützt den Autor darin ein Buch zu schreiben, das auch gelesen wird.

Inhaltsverzeichnis

Urheberrecht

Atomic Kotlin

Von Bruce Eckel, Präsident, MindView, LLC, und Svetlana Isakova, JetBrains sro.

eBook ISBN 978-0-9818725-4-4

Version 1.0: Dezember 2020

Version 1.1: November 2021

Print-Buch ISBN 978-0-9818725-5-1

Erster Druck: Januar 2021

Zweiter Druck: November 2021

Die Aktualisierungen vom November 2021 beinhalten Anpassungen für Kotlin 1.5 und Korrekturen.

Die eBook ISBN deckt die Leanpub und Stepik eBook-Verteilungen ab, beide verfügbar über *www.AtomicKotlin.com*.

Bitte kaufen Sie dieses Buch über *www.AtomicKotlin.com*, um seine fortlaufende Pflege und Aktualisierungen zu unterstützen.

Erstellt in Crested Butte, Colorado, USA, und München, Deutschland.

Text gedruckt in den Vereinigten Staaten.

Umschlaggestaltung von Daniel Will-Harris, www.Will-Harris.com[1]

Besuchen Sie uns auf *www.AtomicKotlin.com*.

Quellcode

Der gesamte Quellcode für dieses Buch ist als urheberrechtlich geschütztes Freeware verfügbar, verteilt über Github[3]. Um sicherzustellen, dass Sie die aktuellste Version haben, ist dies die offizielle Code-Vertriebssite. Sie dürfen diesen Code in Klassenzimmern und anderen Bildungssituationen verwenden, solange Sie dieses Buch als Quelle angeben.

Das Hauptziel dieses Urheberrechts besteht darin, sicherzustellen, dass die Quelle des Codes ordnungsgemäß angegeben wird, und zu verhindern, dass Sie den Code ohne Genehmigung neu veröffentlichen. (Solange dieses Buch zitiert wird, ist die

[1] http://www.Will-Harris.com
[2] https://kotlinlang.org/foundation/kotlin-foundation.html
[3] https://github.com/BruceEckel/AtomicKotlinExamples

Verwendung von Beispielen aus dem Buch in den meisten Medien im Allgemeinen kein Problem.)

In jeder Quellcodedatei finden Sie einen Verweis auf den folgenden Urheberrechtshinweis:

```
// Copyright.txt
This computer source code is Copyright ©2021 MindView LLC.
All Rights Reserved.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "Atomic
Kotlin" is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView LLC, PO Box 969, Crested Butte, CO 81224
MindViewInc@gmail.com

4. The Source Code and documentation are copyrighted by
MindView LLC. The Source code is provided without express
or implied warranty of any kind, including any implied
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView LLC does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView
LLC makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
```

Sie dürfen den Code in Ihren Projekten und im Klassenzimmer verwenden (einschließlich Ihrer Präsentationsmaterialien), solange der Urheberrechtshinweis, der in jeder Quelldatei erscheint, erhalten bleibt.

Abschnitt I: Grundlagen der Programmierung

There was something amazingly enticing about programming—**Vint Cerf**

Dieser Abschnitt ist für Leser gedacht, die gerade das Programmieren lernen. Wenn Sie ein erfahrener Programmierer sind, überspringen Sie diesen Abschnitt und fahren Sie mit Zusammenfassung 1 und Zusammenfassung 2 fort.

Einführung

Dieses Buch ist für engagierte Anfänger und erfahrene Programmierer.

Sie sind ein Anfänger, wenn Sie keine Vorkenntnisse im Programmieren haben, aber "engagiert", weil wir Ihnen gerade genug geben, um es selbst herauszufinden. Wenn Sie fertig sind, haben Sie eine solide Grundlage in der Programmierung und in Kotlin.

Wenn Sie ein erfahrener Programmierer sind, überspringen Sie zu Zusammenfassung 1 und Zusammenfassung 2 und fahren Sie von dort aus fort.

Der "Atomare" Teil des Buchtitels bezieht sich auf Atome als die kleinsten unteilbaren Einheiten. In diesem Buch versuchen wir, nur ein Konzept pro Kapitel einzuführen, sodass die Kapitel nicht weiter unterteilt werden können — daher nennen wir sie *Atome*.

Konzepte

Alle Programmiersprachen bestehen aus Funktionen. Sie wenden diese Funktionen an, um Ergebnisse zu erzielen. Kotlin ist mächtig — es hat nicht nur eine reiche Menge an Funktionen, sondern man kann diese Funktionen normalerweise auf verschiedene Arten ausdrücken.

Wenn alles zu schnell auf Sie einprasselt, könnten Sie denken, Kotlin sei "zu kompliziert".

Dieses Buch versucht, Überforderung zu verhindern. Wir bringen Ihnen die Sprache sorgfältig und gezielt bei, unter Anwendung der folgenden Prinzipien:

1. **Babyschritte und kleine Erfolge**. Wir werfen die Tyrannei des Kapitels ab. Stattdessen präsentieren wir jeden kleinen Schritt als ein *omares Konzept* oder einfach *Atom*, das wie ein winziges Kapitel aussieht. Wir versuchen, pro Atom nur ein neues Konzept vorzustellen. Ein typisches Atom enthält ein oder mehrere kleine, ausführbare Codebeispiele und die erzeugte Ausgabe.

2. **Keine Vorwärtsverweise**. Soweit möglich, vermeiden wir es zu sagen: "Diese Funktionen werden in einem späteren Atom erklärt."
3. **Keine Verweise auf andere Programmiersprachen**. Wir tun dies nur, wenn es notwendig ist. Ein Vergleich mit einer Funktion in einer Sprache, die Sie nicht verstehen, ist nicht hilfreich.
4. **Zeigen, nicht erzählen**. Anstatt eine Funktion verbal zu beschreiben, bevorzugen wir Beispiele und Ausgaben. Es ist besser, eine Funktion im Code zu sehen.
5. **Praxis vor Theorie**. Wir versuchen, zuerst die Mechanik der Sprache zu zeigen und dann zu erklären, warum diese Funktionen existieren. Das ist umgekehrt zur "traditionellen" Lehre, scheint aber oft besser zu funktionieren.

Wenn Sie die Funktionen kennen, können Sie die Bedeutung herausfinden. Es ist in der Regel einfacher, eine einzelne Seite Kotlin zu verstehen als den entsprechenden Code in einer anderen Sprache.

Wo ist der Index?

Dieses Buch ist in Markdown geschrieben und mit Leanpub produziert. Leider unterstützen weder Markdown noch Leanpub Indizes. Indem wir jedoch die kleinstmöglichen Kapitel (Atome) schaffen, die aus einem einzigen Thema pro Atom bestehen, fungiert das Inhaltsverzeichnis als eine Art Index. Darüber hinaus ermöglichen die eBook-Versionen elektronisches Suchen im gesamten Buch.

Querverweise

Ein Verweis auf ein Atom im Buch sieht so aus: Einführung, was in diesem Fall auf das aktuelle Atom verweist. In den verschiedenen eBook-Formaten erzeugt dies einen Hyperlink zu diesem Atom.

Formatierung

In diesem Buch:

- *Kursiv* führt einen neuen Begriff oder ein Konzept ein und betont manchmal eine Idee.
- `Schrift mit fester Breite` zeigt Programmschlüsselwörter, Bezeichner und Dateinamen an. Die Codebeispiele sind ebenfalls in dieser Schriftart und in den eBook-Versionen des Buches farblich hervorgehoben.
- Im Fließtext folgt auf einen Funktionsnamen leere Klammern, wie in `func()`. Dies erinnert den Leser daran, dass er eine Funktion betrachtet.
- Um das eBook auf allen Geräten leicht lesbar zu machen und dem Benutzer zu ermöglichen, die Schriftgröße zu erhöhen, begrenzen wir die Breite unserer Code-Listings auf 47 Zeichen. Dies erfordert manchmal Kompromisse, aber wir glauben, dass die Ergebnisse es wert sind. Um diese Breiten zu erreichen, entfernen wir möglicherweise Leerzeichen, die in vielen Formatierungsstilen ansonsten enthalten wären — insbesondere verwenden wir Einrückungen von zwei Leerzeichen anstelle der standardmäßigen vier Leerzeichen.

“Pause”

Gelegentlich sehen Sie:

- -

Dies zeigt eine Pause oder eine Art kleinen Reset an. In diesem Buch erscheint es oft vor einer kurzen Zusammenfassung des aktuellen Unterabschnitts, wo ein “Zusammenfassung”-Untertitel übertrieben wäre. Einige Bücher verwenden einen Mechanismus wie diesen, um anzuzeigen, dass eine Idee abgeschlossen ist und wir etwas Neues beginnen, das jedoch noch im gleichen Thema liegt und nicht groß genug ist, um einen Unterabschnitt oder einen neuen Abschnitt zu rechtfertigen. Das Markdown in Leanpub ist ziemlich begrenzt, und die Verwendung von einem oder mehreren Punkten (mein ursprünglicher Versuch) ist nicht möglich. Zwei Striche im Markdown zu setzen, erzeugt einen Punkt und einen Strich. Es könnte eine bessere Möglichkeit geben, dies zu tun, aber ich habe sie nicht gefunden, also habe ich mich darauf festgelegt.

Probieren Sie das Buch aus

Wir bieten eine kostenlose Probe des elektronischen Buches auf *AtomicKotlin.com* an. Die Probe enthält die ersten beiden Abschnitte in voller Länge sowie mehrere

nachfolgende Atome. So können Sie das Buch ausprobieren und entscheiden, ob es für Sie geeignet ist.

Das vollständige Buch ist sowohl als Druckversion als auch als eBook erhältlich. Wenn Ihnen gefällt, was wir in der kostenlosen Probe gemacht haben, unterstützen Sie uns bitte und helfen Sie uns, unsere Arbeit fortzusetzen, indem Sie für das bezahlen, was Sie nutzen. Wir hoffen, das Buch hilft Ihnen, und wir schätzen Ihre Unterstützung.

Im Zeitalter des Internets scheint es unmöglich, irgendein Stück Information zu kontrollieren. Sie werden wahrscheinlich die elektronische Version dieses Buches an zahlreichen Orten finden. Wenn Sie im Moment nicht für das Buch zahlen können und es von einer dieser Seiten herunterladen, „geben Sie es bitte weiter". Helfen Sie beispielsweise jemand anderem, die Sprache zu lernen, sobald Sie sie beherrschen. Oder helfen Sie jemandem auf irgendeine Weise, wie er es braucht. Vielleicht geht es Ihnen in Zukunft besser, und dann können Sie für das Buch bezahlen.

Übungen und Lösungen

Die meisten Atome in *Atomic Kotlin* werden von einer Handvoll kleiner Übungen begleitet. Um Ihr Verständnis zu verbessern, empfehlen wir, die Übungen unmittelbar nach dem Lesen des Atoms zu lösen. Die meisten Übungen werden automatisch von der JetBrains IntelliJ IDEA integrierten Entwicklungsumgebung (IDE) überprüft, sodass Sie Ihren Fortschritt sehen und Hinweise erhalten können, wenn Sie feststecken.

Sie finden die folgenden Links unter http://AtomicKotlin.com/exercises/[4].

Um die Übungen zu lösen, installieren Sie IntelliJ IDEA mit dem EduTools-Plugin, indem Sie diesen Tutorials folgen:

1. Installieren Sie IntelliJ IDEA und das EduTools-Plugin[5].
2. Öffnen Sie den Atomic Kotlin-Kurs und lösen Sie die Übungen[6].

Im Kurs finden Sie Lösungen für alle Übungen. Wenn Sie bei einer Übung feststecken, schauen Sie nach Hinweisen oder werfen Sie einen Blick auf die Lösung. Wir empfehlen dennoch, sie selbst zu implementieren.

[4] http://AtomicKotlin.com/exercises/
[5] https://www.jetbrains.com/help/education/install-edutools-plugin.html
[6] https://www.jetbrains.com/help/education/learner-start-guide.html?section=Atomic%20Kotlin

Wenn Sie Probleme bei der Einrichtung und Ausführung des Kurses haben, lesen Sie bitte den Leitfaden zur Fehlerbehebung[7]. Wenn das Ihr Problem nicht löst, wenden Sie sich bitte an das Support-Team, wie im Leitfaden angegeben.

Wenn Sie einen Fehler im Kursinhalt finden (zum Beispiel ein Test für eine Aufgabe liefert das falsche Ergebnis), nutzen Sie bitte unser Issue-Tracker, um das Problem mit diesem vorausgefüllten Formular[8] zu melden. Beachten Sie, dass Sie sich bei YouTrack anmelden müssen. Wir schätzen Ihre Zeit, um den Kurs zu verbessern!

Seminare

Informationen zu Live-Seminaren und anderen Lernwerkzeugen finden Sie auf *AtomicKotlin.com*.

Konferenzen

Bruce organisiert *Open-Spaces*-Konferenzen wie das Winter Tech Forum[9]. Treten Sie der Mailingliste auf *AtomicKotlin.com* bei, um über unsere Aktivitäten und Vorträge informiert zu bleiben.

Unterstützen Sie uns

Dies war ein großes Projekt. Es hat Zeit und Mühe gekostet, dieses Buch und die begleitenden Unterstützungsmaterialien zu erstellen. Wenn Ihnen dieses Buch gefällt und Sie mehr davon sehen möchten, unterstützen Sie uns bitte:

- **Bloggen**, **tweeten Sie**, **usw**. **und erzählen Sie Ihren Freunden davon**. Dies ist eine Graswurzel-Marketing-Bemühung, daher hilft alles, was Sie tun.
- **Kaufen Sie eine eBook- oder Druckversion** dieses Buches auf *AtomicKotlin.com*.
- **Besuchen Sie** *AtomicKotlin.com* für andere Unterstützungsprodukte oder Veranstaltungen.

[7] https://www.jetbrains.com/help/education/troubleshooting-guide.html
[8] https://youtrack.jetbrains.com/newIssue?project=EDC&summary=AtomicKotlin%3A&c=Subsystem%20Kotlin&c=
[9] http://www.WinterTechForum.com

Über uns

Bruce Eckel ist der Autor der mehrfach ausgezeichneten Bücher *Thinking in Java* und *Thinking in C++* sowie einer Reihe weiterer Bücher über Computerprogrammierung, darunter Atomic Scala[10]. Er hat weltweit Hunderte von Präsentationen gehalten und alternative Konferenzen und Veranstaltungen wie das Winter Tech Forum[11] und Entwickler-Retreats organisiert. Bruce hat einen BS in angewandter Physik und einen MS in Computertechnik. Sein Blog befindet sich auf www.BruceEckel.com[12] und sein Beratungs-, Trainings- und Konferenzunternehmen ist Mindview LLC[13].

Svetlana Isakova begann als Mitglied des Kotlin-Compiler-Teams und ist nun eine Entwickler-Botschafterin für JetBrains. Sie unterrichtet Kotlin und spricht auf Konferenzen weltweit und ist Mitautorin des Buches *Kotlin in Action*.

Danksagungen

- Das Kotlin-Sprachdesign-Team und die Mitwirkenden.
- Die Entwickler von Leanpub, die das Veröffentlichen dieses Buches *so* viel einfacher gemacht haben.
- James Ward für die Umwandlung des Gradle-Builds in Kotlin und dafür, dass er im Allgemeinen großartig ist.

Widmungen

Für meinen geliebten Vater, E. Wayne Eckel. 1. April 1924—23. November 2016. Du hast mir zuerst etwas über Maschinen, Werkzeuge und Design beigebracht.

Für meinen Vater, Sergey Lvovich Isakov, der so früh von uns gegangen ist und den wir immer vermissen werden.

[10] http://www.atomicscala.com/
[11] http://www.WinterTechForum.com
[12] http://www.BruceEckel.com
[13] https://www.mindviewllc.com/

Über das Cover

Daniel Will-Harris[14] gestaltete das Cover basierend auf dem Kotlin-Logo.

[14] http://www.will-harris.com

Warum Kotlin?

> *Programme sollen so geschrieben werden, dass Menschen sie lesen können, und erst in zweiter Linie für Maschinen, die sie ausführen.*—**Harold Abelson**, Mitautor, *Structure and Interpretation of Computer Programs.*

Dieses Kapitel bietet einen Überblick über die historische Entwicklung von Programmiersprachen, damit Sie verstehen, wo Kotlin einzuordnen ist und warum Sie es lernen möchten. Wir führen einige Themen ein, die, wenn Sie ein Anfänger sind, momentan zu kompliziert erscheinen mögen. Fühlen Sie sich frei, dieses Kapitel zu überspringen und später darauf zurückzukommen, nachdem Sie mehr vom Buch gelesen haben.

Die Gestaltung von Programmiersprachen ist ein evolutionärer Weg, der von der Erfüllung der Bedürfnisse der Maschine zur Erfüllung der Bedürfnisse des Programmierers führt.

Eine Programmiersprache wird von einem Sprachdesigner erfunden und als eines oder mehrere Programme implementiert, die als Werkzeuge zur Nutzung der Sprache dienen. Der Implementierer ist in der Regel der Sprachdesigner, zumindest anfangs.

Frühe Sprachen konzentrierten sich auf Hardwarebeschränkungen. Mit zunehmender Rechenleistung der Computer verlagerten sich neuere Sprachen hin zu anspruchsvollerer Programmierung mit einem Schwerpunkt auf Zuverlässigkeit. Diese Sprachen können Merkmale basierend auf der Psychologie des Programmierens wählen.

Jede Programmiersprache ist eine Sammlung von Experimenten. Historisch gesehen war das Design von Programmiersprachen eine Abfolge von Vermutungen und Annahmen darüber, was Programmierer produktiver machen könnte. Einige dieser Experimente scheitern, einige sind mäßig erfolgreich und einige sind sehr erfolgreich.

Wir lernen aus den Experimenten jeder neuen Sprache. Einige Sprachen befassen sich mit Problemen, die sich als nebensächlich statt wesentlich erweisen, oder die

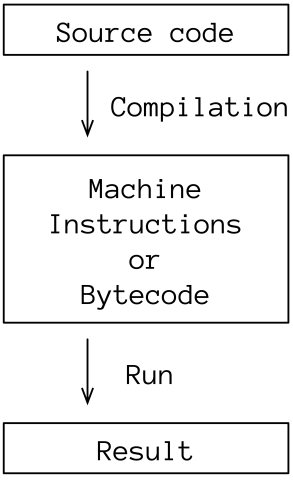
Umgebung ändert sich (schnellere Prozessoren, günstigere Speicher, neues Verständnis von Programmierung und Sprachen) und dieses Problem wird weniger wichtig oder sogar unbedeutend. Wenn diese Ideen veraltet sind und sich die Sprache nicht weiterentwickelt, verschwindet sie aus der Verwendung.

Die ursprünglichen Programmierer arbeiteten direkt mit Zahlen, die Prozessor-Maschinenbefehle darstellten. Dieser Ansatz führte zu zahlreichen Fehlern, und *Assemblersprache* wurde geschaffen, um die Zahlen durch mnemonische *Opcodes*—Wörter, die sich Programmierer leichter merken und lesen konnten, zusammen mit anderen hilfreichen Werkzeugen zu ersetzen. Es gab jedoch immer noch eine Eins-zu-eins-Entsprechung zwischen Assemblerbefehlen und Maschinenbefehlen, und Programmierer mussten jede Zeile Assemblercode schreiben. Darüber hinaus verwendete jeder Computerprozessor seine eigene spezifische Assemblersprache.

Das Entwickeln von Programmen in Assemblersprache ist äußerst kostspielig. Höhere Programmiersprachen helfen, dieses Problem zu lösen, indem sie eine Abstraktionsebene von den niedrigeren Assemblersprachen schaffen.

Compiler und Interpreter

Die Anweisungen einer interpretierten Sprache werden direkt von einem Programm namens *Interpreter* ausgeführt. Kotlin wird *kompiliert* statt *interpretiert*. Der Quellcode einer kompilierten Sprache wird in eine andere Darstellung umgewandelt, die als eigenes Programm läuft, entweder direkt auf einem Hardwareprozessor oder auf einer *virtuellen Maschine*, die einen Prozessor emuliert:



Sprachen wie C, C++, Go und Rust werden in *Maschinencode* kompiliert, der direkt auf der zugrunde liegenden Hardware-*Zentralverarbeitungseinheit* (CPU) läuft. Sprachen wie Java und Kotlin werden in *Bytecode* kompiliert, der ein Zwischenformat ist, das nicht direkt auf der Hardware-CPU läuft, sondern auf einer *virtuellen Maschine*, einem Programm, das Bytecode-Anweisungen ausführt. Programme, die von der JVM-Version von Kotlin erzeugt werden, laufen auf der *Java Virtual Machine* (JVM).

Die Portabilität ist ein wichtiger Vorteil einer virtuellen Maschine. Der gleiche Bytecode kann auf jedem Computer laufen, der eine virtuelle Maschine hat. Virtuelle Maschinen können für spezielle Hardware optimiert werden und Geschwindigkeitsprobleme lösen. Die JVM enthält viele Jahre solcher Optimierungen und wurde auf vielen Plattformen implementiert.

Zur *Kompilierungszeit* wird der Code vom Compiler überprüft, um *Kompilierungsfehler* zu entdecken. (IntelliJ IDEA und andere Entwicklungsumgebungen heben diese Fehler hervor, wenn Sie den Code eingeben, sodass Sie schnell Probleme entdecken und beheben können). Wenn es keine Kompilierungsfehler gibt, wird der Quellcode in Bytecode kompiliert.

Ein *Laufzeitfehler* kann zur Kompilierungszeit nicht entdeckt werden, daher tritt er erst auf, wenn Sie das Programm ausführen. Typischerweise sind Laufzeitfehler schwieriger zu entdecken und teurer zu beheben. *Statisch typisierte Sprachen* wie Kotlin entdecken so viele Fehler wie möglich zur Kompilierungszeit, während

dynamische Sprachen ihre Sicherheitsprüfungen zur Laufzeit durchführen (einige dynamische Sprachen führen nicht so viele Sicherheitsprüfungen durch, wie sie könnten).

Sprachen, die Kotlin beeinflusst haben

Kotlin zieht seine Ideen und Merkmale aus vielen Sprachen, und diese Sprachen wurden von früheren Sprachen beeinflusst. Es ist hilfreich, etwas über die Geschichte der Programmiersprachen zu wissen, um Perspektiven zu gewinnen, wie wir zu Kotlin gekommen sind. Die hier beschriebenen Sprachen wurden wegen ihres Einflusses auf die nachfolgenden Sprachen ausgewählt. All diese Sprachen inspirierten letztendlich das Design von Kotlin, manchmal indem sie ein Beispiel dafür waren, was man *nicht* tun sollte.

FORTRAN: FORmula TRANslation (1957)

Entwickelt für den Einsatz durch Wissenschaftler und Ingenieure, war das Ziel von Fortran, das Codieren von Gleichungen zu erleichtern. Fein abgestimmte und getestete Fortran-Bibliotheken sind noch heute im Einsatz, werden jedoch typischerweise "umwickelt", um sie von anderen Sprachen aus aufrufbar zu machen.

LISP: LISt Processor (1958)

Anstatt anwendungsspezifisch zu sein, verkörperte LISP wesentliche Programmierkonzepte; es war die Sprache der Informatiker und die erste *funktionale* Programmiersprache (Sie werden in diesem Buch über funktionale Programmierung lernen). Der Kompromiss für seine Macht und Flexibilität war die Effizienz: LISP war typischerweise zu teuer, um auf frühen Maschinen ausgeführt zu werden, und erst in den letzten Jahrzehnten wurden Maschinen schnell genug, um eine Wiederbelebung der Nutzung von LISP zu ermöglichen. Zum Beispiel ist der GNU Emacs-Editor vollständig in LISP geschrieben und kann mit LISP erweitert werden.

ALGOL: ALGOrithmic Language (1958)

Wahrscheinlich die einflussreichste der Sprachen der 1950er Jahre, da sie eine Syntax einführte, die in vielen nachfolgenden Sprachen Bestand hatte. Zum Beispiel sind C und seine Derivate "ALGOL-ähnliche" Sprachen.

COBOL: COmmon Business-Oriented Language (1959)

Entwickelt für Geschäft, Finanzen und administrative Datenverarbeitung. Es hat eine englischartige Syntax und sollte selbstdokumentierend und sehr lesbar sein. Obwohl diese Absicht im Allgemeinen scheiterte—COBOL ist berüchtigt für Fehler, die durch ein fehlplatziertes Punktzeichen eingeführt wurden—zwang das US-Verteidigungsministerium die weitverbreitete Einführung auf Großrechnern, und Systeme laufen (und erfordern Wartung) noch heute.

BASIC: Beginners' All-purpose Symbolic Instruction Code (1964)

BASIC war einer der frühen Versuche, Programmieren zugänglich zu machen. Obwohl sehr erfolgreich, waren seine Funktionen und Syntax begrenzt, sodass es nur teilweise hilfreich für Menschen war, die anspruchsvollere Sprachen lernen mussten. Es ist überwiegend eine interpretierte Sprache, was bedeutet, dass man den ursprünglichen Code für das Programm benötigt, um es auszuführen. Trotzdem wurden viele nützliche Programme in BASIC geschrieben, insbesondere als Skriptsprache für Microsofts "Office"-Produkte. BASIC könnte sogar als die erste "offene" Programmiersprache betrachtet werden, da zahlreiche Variationen davon erstellt wurden.

Simula 67, die ursprüngliche objektorientierte Sprache (1967)

Eine *Simulation* beinhaltet typischerweise viele "Objekte", die miteinander interagieren. Verschiedene Objekte haben unterschiedliche Eigenschaften und Verhaltensweisen. Die zu der Zeit existierenden Sprachen waren unhandlich für Simulationen zu

verwenden, daher wurde Simula (eine weitere "ALGOL-ähnliche" Sprache) entwickelt, um direkte Unterstützung für die Erstellung von Simulationsobjekten zu bieten. Es stellt sich heraus, dass diese Ideen auch für allgemeine Programmierung nützlich sind, und dies war der Ursprung der objektorientierten (OO) Sprachen.

Pascal (1970)

Pascal erhöhte die Kompilierungsgeschwindigkeit, indem es die Sprache so einschränkte, dass sie als *Einzelpass-Compiler* implementiert werden konnte. Die Sprache zwang den Programmierer, ihren Code auf eine bestimmte Weise zu strukturieren und legte etwas umständliche und weniger lesbare Einschränkungen für die Programmorganisation auf. Da Prozessoren schneller wurden, Speicher billiger und die Compiler-Technologie besser, wurden die Auswirkungen dieser Einschränkungen zu kostspielig.

Eine Implementierung von Pascal, Turbo Pascal von Borland, arbeitete zunächst auf CP/M-Maschinen und machte dann den Sprung zu frühen MS-DOS (Vorläufer von Windows), später entwickelte es sich zur Delphi-Sprache für Windows. Indem alles im Speicher untergebracht wurde, kompilierte Turbo Pascal in atemberaubender Geschwindigkeit auf sehr leistungsschwachen Maschinen, was das Programmerlebnis dramatisch verbesserte. Sein Schöpfer, Anders Hejlsberg, entwarf später sowohl C# als auch TypeScript.

Niklaus Wirth, der Erfinder von Pascal, schuf nachfolgende Sprachen: Modula, Modula-2 und Oberon. Wie der Name schon sagt, konzentrierte sich Modula auf die Aufteilung von Programmen in Module, für bessere Organisation und schnellere Kompilierung. Die meisten modernen Sprachen unterstützen *separate Kompilierung* und eine Form von Modulsystem.

C (1972)

Trotz der zunehmenden Zahl von Hochsprachen schrieben Programmierer immer noch in Assemblersprache. Dies wird oft als *Systemprogrammierung* bezeichnet, da es auf Ebene des Betriebssystems erfolgt, umfasst aber auch eingebettete Programmierung für spezielle physische Geräte. Dies ist nicht nur mühsam und teuer (Bruce begann seine Karriere mit dem Schreiben von Assemblersprache für eingebettete

Systeme), sondern es ist auch nicht portabel—Assemblersprache kann nur auf dem Prozessor laufen, für den sie geschrieben wurde. C wurde als "hochlevelige Assemblersprache" entworfen, die dennoch nah genug an der Hardware ist, dass man selten Assemblersprache schreiben muss. Noch wichtiger ist, dass ein C-Programm auf jedem Prozessor mit einem C-Compiler läuft. C entkoppelte das Programm vom Prozessor, was ein großes und teures Problem löste. Als Ergebnis konnten ehemalige Assemblersprachen-Programmierer in C weitaus produktiver sein. C war so effektiv, dass neuere Sprachen (insbesondere Go und Rust) immer noch versuchen, es für die Systemprogrammierung abzulösen.

Smalltalk (1972)

Von Anfang an als rein objektorientiert konzipiert, hat Smalltalk die OO- und Sprachtheorie erheblich vorangebracht, indem es eine Plattform für Experimente war und die schnelle Anwendungsentwicklung demonstrierte. Es wurde jedoch in einer Zeit entwickelt, als Sprachen noch proprietär waren, und der Einstiegspreis für ein Smalltalk-System konnte in die Tausende gehen. Es war interpretiert, sodass man eine Smalltalk-Umgebung benötigte, um Programme auszuführen. Open-Source-Smalltalk-Implementierungen erschienen erst, nachdem die Programmierwelt sich weiterentwickelt hatte. Smalltalk-Programmierer haben großartige Einblicke geliefert, die späteren OO-Sprachen wie C++ und Java zugutekamen.

C++: Ein besseres C mit Objekten (1983)

Bjarne Stroustrup schuf C++, weil er ein besseres C wollte und Unterstützung für die objektorientierten Konstrukte, die er bei der Verwendung von Simula-67 erlebt hatte. Bruce war acht Jahre lang Mitglied des C++-Normungsausschusses und schrieb drei Bücher über C++, darunter *Thinking in C++*.

Rückwärtskompatibilität mit C war ein grundlegendes Prinzip des C++-Designs, sodass C-Code in C++ mit praktisch keinen Änderungen kompiliert werden kann. Dies bot einen einfachen Migrationspfad - Programmierer konnten weiterhin in C programmieren, die Vorteile von C++ nutzen und langsam mit C++-Funktionen experimentieren, während sie produktiv blieben. Die meisten Kritiken an C++ lassen sich auf die Einschränkung der Rückwärtskompatibilität mit C zurückführen.

Eines der Probleme bei C war das Thema *Speicherverwaltung*. Der Programmierer muss zuerst Speicher erwerben, dann eine Operation mit diesem Speicher ausführen und dann den Speicher freigeben. Das Vergessen, Speicher freizugeben, wird als *Speicherleck* bezeichnet und kann dazu führen, dass der verfügbare Speicher aufgebraucht wird und der Prozess abstürzt. Die anfängliche Version von C++ machte einige Innovationen in diesem Bereich, zusammen mit *Konstruktoren*, um eine ordnungsgemäße Initialisierung sicherzustellen. Spätere Versionen der Sprache haben bedeutende Verbesserungen in der Speicherverwaltung vorgenommen.

Python: Freundlich und flexibel (1990)

Der Designer von Python, Guido Van Rossum, schuf die Sprache basierend auf seiner Inspiration des „Programmieren für alle". Seine Pflege der Python-Community hat ihr den Ruf verliehen, die freundlichste und unterstützendste Community in der Programmierwelt zu sein. Python war eine der ersten Open-Source-Sprachen, was zu Implementierungen auf praktisch jeder Plattform führte, einschließlich eingebetteter Systeme und maschinellem Lernen. Seine Dynamik und Benutzerfreundlichkeit machen es ideal für die Automatisierung kleiner, sich wiederholender Aufgaben, aber seine Funktionen unterstützen auch die Erstellung großer, komplexer Programme.

Python ist eine echte "Grassroots"-Sprache; es hatte nie ein Unternehmen, das es förderte, und die Einstellung seiner Fans war, die Sprache niemals zu pushen, sondern einfach jedem zu helfen, der sie lernen möchte. Die Sprache verbessert sich stetig, und in den letzten Jahren ist ihre Popularität explodiert.

Python könnte die erste Mainstream-Sprache gewesen sein, die funktionale und OO-Programmierung kombinierte. Es war Java voraus mit automatischer Speicherverwaltung durch *Müllabfuhr* (normalerweise müssen Sie selbst nie Speicher zuweisen oder freigeben) und der Fähigkeit, Programme auf mehreren Plattformen auszuführen.

Haskell: Reine funktionale Programmierung (1990)

Inspiriert von Miranda (1985), einer proprietären Sprache, wurde Haskell als offener Standard für die Forschung zur reinen funktionalen Programmierung geschaffen, obwohl es auch für Produkte verwendet wurde. Syntax und Ideen von Haskell haben eine Reihe nachfolgender Sprachen beeinflusst, darunter Kotlin.

Java: Virtuelle Maschinen und Müllabfuhr (1995)

James Gosling und sein Team erhielten die Aufgabe, Code für eine TV-Set-Top-Box zu schreiben. Sie entschieden, dass sie C++ nicht mochten und anstatt die Box zu erstellen, entwickelten sie die Java-Sprache. Das Unternehmen, Sun Microsystems, setzte einen enormen Marketingdruck hinter die kostenlose Sprache (damals eine neue Idee), um die aufkommende Internetlandschaft zu dominieren.

Dieses wahrgenommene Zeitfenster für die Internet-Dominanz setzte das Java-Sprachdesign unter erheblichen Druck, was zu einer beträchtlichen Anzahl von Mängeln führte (Das Buch *Thinking in Java* beleuchtet diese Mängel, damit die Leser darauf vorbereitet sind, mit ihnen umzugehen). Brian Goetz bei Oracle, der derzeitige leitende Entwickler von Java, hat bemerkenswerte und überraschende Verbesserungen an Java vorgenommen, trotz der Einschränkungen, die er geerbt hat. Obwohl Java bemerkenswert erfolgreich war, ist ein wichtiges Kotlin-Designziel, die Mängel von Java zu beheben, damit Programmierer produktiver sein können.

Der Erfolg von Java beruht auf zwei innovativen Funktionen: einer *virtuellen Maschine* und *Müllabfuhr*. Diese waren in anderen Sprachen verfügbar - zum Beispiel haben LISP, Smalltalk und Python Müllabfuhr, und UCSD Pascal lief auf einer virtuellen Maschine -, aber sie wurden nie als praktikabel für Mainstream-Sprachen angesehen. Java änderte das und machte Programmierer dadurch erheblich produktiver.

Eine virtuelle Maschine ist eine Zwischenebene zwischen der Sprache und der Hardware. Die Sprache muss keinen Maschinencode für einen bestimmten Prozessor erzeugen; sie muss nur eine Zwischen-Sprache (Bytecode) erzeugen, die auf der virtuellen Maschine läuft. Virtuelle Maschinen erfordern Rechenleistung und wurden vor Java als unpraktisch angesehen. Die *Java Virtual Machine* (JVM) führte zu Javas Slogan "write once, run everywhere." Darüber hinaus können andere Sprachen leichter entwickelt werden, indem sie die JVM anvisieren; Beispiele umfassen Groovy, eine Java-ähnliche Skriptsprache, und Clojure, eine Version von LISP.

Die Müllabfuhr löst das Problem, das Freigeben von Speicher zu vergessen, oder wenn es schwierig ist, zu wissen, wann ein Speicherplatz nicht mehr genutzt wird. Projekte wurden erheblich verzögert oder sogar abgebrochen wegen Speicherlecks. Obwohl die Müllabfuhr in einigen früheren Sprachen vorkommt, galt sie als inakzeptabel aufwändig, bis Java ihre Praktikabilität demonstrierte.

JavaScript: Nur dem Namen nach Java (1995)

Der ursprüngliche Webbrowser kopierte und zeigte einfach Seiten von einem Webserver an. Webbrowser vervielfältigten sich und wurden zu einer neuen Programmierplattform, die Sprachunterstützung benötigte. Java wollte diese Sprache sein, war aber zu umständlich für den Job. JavaScript begann als LiveScript und wurde in NetScape Navigator integriert, einen der ersten Webbrowser. Die Umbenennung in JavaScript war ein Marketingtrick von NetScape, da die Sprache nur eine vage Ähnlichkeit mit Java hat.

Als das Web aufblühte, wurde JavaScript enorm wichtig. Das Verhalten von JavaScript war jedoch so unvorhersehbar, dass Douglas Crockford ein Buch mit dem ironischen Titel *JavaScript, the Good Parts* schrieb, in dem er alle Probleme mit der Sprache aufzeigte, damit Programmierer sie vermeiden können. Nachfolgende Verbesserungen durch das ECMAScript-Komitee haben JavaScript für einen ursprünglichen JavaScript-Programmierer unkenntlich gemacht. Es wird jetzt als stabile und ausgereifte Sprache betrachtet.

Web-Assembly (WASM) wurde von JavaScript abgeleitet, um eine Art Bytecode für Webbrowser zu sein. Es läuft oft viel schneller als JavaScript und kann von anderen Sprachen generiert werden. Zum Zeitpunkt des Schreibens arbeitet das Kotlin-Team daran, WASM als Ziel hinzuzufügen.

C#: Java für .NET (2000)

C# wurde entwickelt, um einige der wichtigen Fähigkeiten von Java auf der .NET (Windows) Plattform bereitzustellen, während es den Designern freistellte, sich nicht an die Java-Sprache zu halten. Das Ergebnis beinhaltete zahlreiche Verbesserungen gegenüber Java. Zum Beispiel entwickelte C# das Konzept der *Erweiterungsfunktionen*, die in Kotlin stark genutzt werden. C# wurde auch deutlich funktionaler als Java. Viele C#-Funktionen haben offensichtlich das Design von Kotlin beeinflusst.

Scala: SCALAble (2003)

Martin Odersky schuf Scala, um auf der Java Virtual Machine zu laufen: Um auf der Arbeit auf der JVM aufzubauen, um mit Java-Programmen zu interagieren

und möglicherweise mit der Idee, dass es Java verdrängen könnte. Als Forscher nutzten Odersky und sein Team Scala als Plattform, um mit Sprachmerkmalen zu experimentieren, insbesondere solchen, die nicht in Java enthalten sind.

Diese Experimente waren erhellend, und eine Reihe davon fand in modifizierter Form ihren Weg nach Kotlin. Zum Beispiel wird die Fähigkeit, Operatoren wie + für spezielle Fälle neu zu definieren, als *Operatorüberladung* bezeichnet. Dies war in C++ enthalten, aber nicht in Java. Scala fügte die Operatorüberladung hinzu, erlaubt aber auch die Erfindung neuer Operatoren durch Kombination beliebiger Zeichenfolgen. Dies führt oft zu verwirrenderen Code. Eine begrenzte Form der Operatorüberladung ist in Kotlin enthalten, aber man kann nur Operatoren überladen, die bereits existieren.

Scala ist auch ein objekt-funktionales Hybrid, ähnlich wie Python, aber mit einem Fokus auf reine Funktionen und strenge Objekte. Dies inspirierte die Entscheidung von Kotlin, ebenfalls ein objekt-funktionales Hybrid zu sein.

Wie Scala läuft Kotlin auf der JVM, interagiert jedoch viel einfacher mit Java als Scala (siehe Anhang B). Darüber hinaus zielt Kotlin auf JavaScript, das Android-Betriebssystem und erzeugt nativen Code für andere Plattformen.

Atomic Kotlin entwickelte sich aus den Ideen und Materialien von Atomic Scala[15].

Groovy: Eine dynamische JVM-Sprache (2007)

Dynamische Sprachen sind ansprechend, weil sie interaktiver und prägnanter sind als statische Sprachen. Es gab zahlreiche Versuche, ein dynamischeres Programmiererlebnis auf der JVM zu erreichen, darunter Jython (Python) und Clojure (ein Dialekt von Lisp). Groovy war die erste, die breite Akzeptanz erreichte.

Auf den ersten Blick erscheint Groovy als bereinigte Version von Java, die ein angenehmeres Programmiererlebnis bietet. Der meiste Java-Code läuft unverändert in Groovy, sodass Java-Programmierer schnell produktiv sein können und später die anspruchsvolleren Funktionen erlernen können, die bemerkenswerte Programmierverbesserungen gegenüber Java bieten.

Die Kotlin-Operatoren `?.` und `?:`, die sich mit dem Problem der Leere beschäftigen, erschienen zuerst in Groovy.

[15] http://www.AtomicScala.com

Es gibt zahlreiche Groovy-Funktionen, die in Kotlin erkennbar sind. Einige dieser Funktionen erscheinen auch in anderen Sprachen, was wahrscheinlich stärker dafür drängte, dass sie in Kotlin aufgenommen wurden.

Warum Kotlin? (Eingeführt 2011, Version 1.0: 2016)

Genauso wie C++ ursprünglich als "ein besseres C" gedacht war, war Kotlin zunächst darauf ausgerichtet, "ein besseres Java" zu sein. Es hat sich seitdem erheblich über dieses Ziel hinaus entwickelt.

Kotlin wählt pragmatisch nur die erfolgreichsten und hilfreichsten Funktionen aus anderen Programmiersprachen aus—nachdem diese Funktionen in der Praxis getestet und als besonders wertvoll erwiesen wurden.

Wenn Sie also von einer anderen Sprache kommen, könnten Sie einige Funktionen dieser Sprache in Kotlin wiedererkennen. Dies ist beabsichtigt: Kotlin maximiert die Produktivität, indem es bewährte Konzepte nutzt.

Lesbarkeit

Lesbarkeit ist ein Hauptziel bei der Gestaltung der Sprache. Die Kotlin-Syntax ist prägnant—sie erfordert in den meisten Szenarien keine Förmlichkeit, kann aber dennoch komplexe Ideen ausdrücken.

Werkzeuge

Kotlin stammt von JetBrains, einem Unternehmen, das sich auf Entwicklerwerkzeuge spezialisiert hat. Es bietet erstklassige Unterstützung für Werkzeuge, und viele Sprachmerkmale wurden mit Blick auf Werkzeuge entwickelt.

Multi-Paradigma

Kotlin unterstützt mehrere Programmierparadigmen, die in diesem Buch sanft eingeführt werden:

- Imperatives Programmieren
- Funktionales Programmieren
- Objektorientiertes Programmieren

Multi-Plattform

Kotlin-Quellcode kann in verschiedene Zielplattformen kompiliert werden:

- **JVM**. Der Quellcode wird in JVM-Bytecode (`.class` Dateien) kompiliert, der dann auf jeder Java Virtual Machine (JVM) ausgeführt werden kann.
- **Android**. Android hat seine eigene Laufzeitumgebung namens ART[16] (der Vorgänger hieß Dalvik). Der Kotlin-Quellcode wird in das *Dalvik Executable Format* (`.dex` Dateien) kompiliert.
- **JavaScript**, um innerhalb eines Webbrowsers ausgeführt zu werden.
- **Native Binaries** durch die Generierung von Maschinencode für spezifische Plattformen und CPUs.

Dieses Buch konzentriert sich auf die Sprache selbst, wobei die JVM als einzige Zielplattform verwendet wird. Sobald Sie die Sprache beherrschen, können Sie Kotlin auf verschiedene Anwendungen und Zielplattformen anwenden.

Zwei Kotlin-Funktionen

Dieses Atom setzt nicht voraus, dass Sie ein Programmierer sind, was es schwierig macht, die meisten Vorteile von Kotlin gegenüber Alternativen zu erklären. Es gibt jedoch zwei Themen, die sehr wirkungsvoll sind und zu diesem frühen Zeitpunkt erklärt werden können: Java-Interoperabilität und das Problem, "keinen Wert" anzuzeigen.

Mühelose Java-Interoperabilität

Um "ein besseres C" zu sein, muss C++ rückwärtskompatibel mit der Syntax von C sein, aber Kotlin muss nicht rückwärtskompatibel mit der Syntax von Java sein—es

[16] https://source.android.com/devices/tech/dalvik

muss nur mit der JVM arbeiten. Dies befreit die Kotlin-Designer, eine viel sauberere und leistungsfähigere Syntax zu schaffen, ohne das visuelle Rauschen und die Komplikationen, die Java überladen.

Damit Kotlin "ein besseres Java" ist, muss das Erlebnis, es auszuprobieren, angenehm und reibungslos sein, sodass Kotlin eine mühelose Integration mit bestehenden Java-Projekten ermöglicht. Sie können ein kleines Stück Kotlin-Funktionalität schreiben und es mitten in Ihren bestehenden Java-Code einfügen. Der Java-Code merkt nicht einmal, dass der Kotlin-Code da ist—er sieht einfach wie weiterer Java-Code aus.

Unternehmen untersuchen oft eine neue Sprache, indem sie ein eigenständiges Programm mit dieser Sprache erstellen. Idealerweise ist dieses Programm nützlich, aber nicht essentiell, sodass es bei einem Scheitern des Projekts mit minimalem Schaden beendet werden kann. Nicht jedes Unternehmen möchte die Ressourcen aufwenden, die für diese Art von Experimenten erforderlich sind. Da Kotlin sich nahtlos in ein bestehendes Java-System integriert (und von dessen Tests profitiert), wird es sehr billig oder sogar kostenlos, Kotlin auszuprobieren, um zu sehen, ob es passt.

Darüber hinaus bietet JetBrains, das Unternehmen, das Kotlin erstellt, IntelliJ IDEA in einer "Community" (kostenlosen) Version an, die Unterstützung sowohl für Java als auch für Kotlin beinhaltet und die Möglichkeit bietet, die beiden einfach zu integrieren. Es gibt sogar ein Tool, das Java-Code nimmt und ihn (größtenteils) in Kotlin umschreibt.

Anhang B behandelt Java-Interoperabilität.

Darstellung von Leere

Ein besonders vorteilhaftes Kotlin-Feature ist seine Lösung für ein herausforderndes Programmierproblem.

Was tun Sie, wenn Ihnen jemand ein Wörterbuch in die Hand drückt und Sie bittet, ein Wort nachzuschlagen, das nicht existiert? Sie könnten Ergebnisse garantieren, indem Sie Definitionen für unbekannte Wörter erfinden. Ein nützlicherer Ansatz ist einfach zu sagen: "Es gibt keine Definition für dieses Wort." Dies zeigt ein erhebliches Problem in der Programmierung: Wie zeigt man "keinen Wert" für einen Speicherplatz an, der nicht initialisiert ist, oder für das Ergebnis einer Operation?

Die *Null-Referenz* wurde 1965 für ALGOL von Tony Hoare erfunden, der sie später als "meinen Milliarden-Dollar-Fehler" bezeichnete. Ein Problem war, dass sie zu einfach war—manchmal reicht es nicht aus, zu wissen, dass ein Raum leer ist. Man muss vielleicht wissen, *warum* er leer ist. Dies führt zum zweiten Problem: der Implementierung. Aus Effizienzgründen war es typischerweise nur ein spezieller Wert, der in eine kleine Menge Speicher passte, und was war besser als der Speicher, der bereits für diese Information bereitgestellt war?

Die ursprüngliche C-Sprache initialisierte den Speicher nicht automatisch, was zahlreiche Probleme verursachte. C++ verbesserte die Situation, indem neu zugewiesener Speicher auf null gesetzt wurde. Wenn also ein numerischer Wert nicht initialisiert ist, ist er einfach eine numerische Null. Das schien nicht so schlimm zu sein, aber es ermöglichte es, dass nicht initialisierte Werte unbemerkt durchrutschten (neuere C- und C++-Compiler warnen oft davor). Schlimmer noch, wenn ein Speicherstück ein *Zeiger* war—verwendet, um auf ein anderes Speicherstück zu verweisen—würde ein Nullzeiger auf die Speicheradresse null zeigen, was fast sicher nicht das ist, was man will.

Java verhindert Zugriffe auf nicht initialisierte Werte, indem es solche Fehler zur Laufzeit meldet. Obwohl dies nicht initialisierte Werte entdeckt, löst es das Problem nicht, denn die einzige Möglichkeit, zu überprüfen, ob Ihr Programm nicht abstürzt, besteht darin, es auszuführen. Es gibt Schwärme dieser Art von Fehlern im Java-Code, und Programmierer verschwenden enorme Mengen an Zeit, um sie zu finden.

Kotlin löst dieses Problem, indem es Operationen verhindert, die Nullfehler verursachen könnten, zur Kompilierzeit, *bevor das Programm ausgeführt werden kann.* Dies ist das am meisten gefeierte Merkmal von Java-Programmierern, die Kotlin übernehmen. Diese eine Funktion kann Java's Nullfehler minimieren oder eliminieren, was Ihrem Projekt erhebliche Mengen an Zeit und Geld spart.

Eine Fülle von Vorteilen

Die beiden Funktionen, die wir hier erklären konnten (ohne mehr Programmierkenntnisse zu erfordern), machen einen großen Unterschied, unabhängig davon, ob Sie ein Java-Programmierer sind oder nicht. Wenn Kotlin Ihre erste Sprache ist und Sie an einem Projekt arbeiten, das mehr Programmierer benötigt, ist es viel einfacher, einen der vielen existierenden Java-Programmierer für Kotlin zu gewinnen.

Kotlin hat viele weitere Vorteile, die wir erst erklären können, wenn Sie mehr über das Programmieren wissen. Dafür ist der Rest des Buches da.

- -

Sprachen werden oft aus Leidenschaft gewählt, nicht aus Vernunft... Ich versuche, Kotlin zu einer Sprache zu machen, die aus einem Grund geliebt wird.—Andrey Breslav, Kotlin Lead Language Designer.

Hallo, Welt!

> "Hello, world!" ist ein Programm, das häufig verwendet wird, um die grundlegende Syntax von Programmiersprachen zu demonstrieren.

Wir entwickeln dieses Programm in mehreren Schritten, damit Sie seine Teile verstehen.

Zuerst lassen Sie uns ein leeres Programm untersuchen, das überhaupt nichts tut:

```
// HelloWorld/EmptyProgram.kt

fun main() {
  // Program code here ...
}
```

Das Beispiel beginnt mit einem *Kommentar*, der ein erläuternder Text ist, der von Kotlin ignoriert wird. `//` (zwei Schrägstriche) leitet einen Kommentar ein, der bis zum Ende der aktuellen Zeile geht:

```
// Single-line comment
```

Kotlin ignoriert das `//` und alles danach bis zum Ende der Zeile. In der folgenden Zeile wird es wieder beachtet.

Die erste Zeile jedes Beispiels in diesem Buch ist ein Kommentar, der mit dem Namen des Unterverzeichnisses beginnt, das die Quellcodedatei enthält (hier `HelloWorld`), gefolgt vom Namen der Datei: `EmptyProgram.kt`. Das Beispiel-Unterverzeichnis für jedes Atom entspricht dem Namen dieses Atoms.

Schlüsselwörter sind von der Sprache reserviert und haben eine spezielle Bedeutung. Das Schlüsselwort `fun` steht für *Funktion*. Eine Funktion ist eine Sammlung von Code, die unter Verwendung des Namens dieser Funktion ausgeführt werden kann (wir verbringen viel Zeit im Buch mit Funktionen). Der Name der Funktion folgt dem `fun` Schlüsselwort, in diesem Fall ist es `main()` (im Fließtext folgen wir dem Funktionsnamen mit Klammern).

`main()` ist tatsächlich ein spezieller Name für eine Funktion; es zeigt den "Einstiegspunkt" für ein Kotlin-Programm an. Ein Kotlin-Programm kann viele Funktionen mit verschiedenen Namen haben, aber `main()` ist diejenige, die automatisch aufgerufen wird, wenn Sie das Programm ausführen.

Die *Parameterliste* folgt dem Funktionsnamen und ist in Klammern eingeschlossen. Hier übergeben wir nichts an `main()`, daher ist die Parameterliste leer.

Der *Funktionskörper* erscheint nach der Parameterliste. Er beginnt mit einer öffnenden Klammer ({) und endet mit einer schließenden Klammer (}). Der Funktionskörper enthält *Anweisungen* und *Ausdrücke*. Eine Anweisung erzeugt eine Wirkung, und ein Ausdruck liefert ein Ergebnis.

`EmptyProgram.kt` enthält keine Anweisungen oder Ausdrücke im Körper, nur einen Kommentar.

Lassen Sie das Programm "Hello, world!" anzeigen, indem Sie eine Zeile im `main()`-Körper hinzufügen:

```
// HelloWorld/HelloWorld.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

Die Zeile, die die Begrüßung anzeigt, beginnt mit `println()`. Wie `main()`, ist `println()` eine Funktion. Diese Zeile *ruft* die Funktion auf, die dann ihren Körper ausführt. Man gibt den Funktionsnamen an, gefolgt von Klammern, die einen oder mehrere Parameter enthalten. In diesem Buch fügen wir beim Bezug auf eine Funktion in der Prosa nach dem Namen Klammern hinzu, um daran zu erinnern, dass es sich um eine Funktion handelt. Hier sagen wir `println()`.

`println()` nimmt einen einzelnen Parameter, der ein `String` ist. Ein `String` wird definiert, indem man Zeichen in Anführungszeichen setzt.

`println()` bewegt den Cursor nach der Anzeige seines Parameters in eine neue Zeile, sodass nachfolgende Ausgaben in der nächsten Zeile erscheinen. Man kann stattdessen `print()` verwenden, das den Cursor in derselben Zeile belässt.

Im Gegensatz zu einigen Sprachen benötigt man in Kotlin kein Semikolon am Ende eines Ausdrucks. Es ist nur notwendig, wenn man mehr als einen Ausdruck auf eine einzelne Zeile setzt (was nicht empfohlen wird).

Für einige Beispiele im Buch zeigen wir die Ausgabe am Ende der Auflistung in einem *mehrzeiligen Kommentar*. Ein mehrzeiliger Kommentar beginnt mit einem `/*` (einem Schrägstrich gefolgt von einem Sternchen) und setzt sich fort – einschließlich Zeilenumbrüchen (die wir *neue Zeilen* nennen) – bis ein `*/` (ein Sternchen gefolgt von einem Schrägstrich) den Kommentar beendet:

```
/* A multiline comment
Doesn't care
about newlines */
```

Es ist möglich, Code auf derselben Zeile *nach* dem schließenden `*/` eines Kommentars hinzuzufügen, aber das ist verwirrend, daher tun es die Leute normalerweise nicht.

Kommentare fügen Informationen hinzu, die nicht offensichtlich aus dem Code hervorgehen. Wenn Kommentare nur wiederholen, was der Code sagt, werden sie lästig und die Leute beginnen, sie zu ignorieren. Wenn sich der Code ändert, vergessen Programmierer oft, die Kommentare zu aktualisieren, daher ist es eine gute Praxis, Kommentare sparsam zu verwenden, hauptsächlich um knifflige Aspekte Ihres Codes hervorzuheben.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

`var` & `val`

> Wenn ein Bezeichner Daten enthält, müssen Sie entscheiden, ob er neu zugewiesen werden kann.

Sie erstellen *Bezeichner*, um auf Elemente in Ihrem Programm zu verweisen. Die grundlegendste Entscheidung für einen Datenbezeichner ist, ob er seinen Inhalt während der Programmausführung ändern kann oder ob er nur einmal zugewiesen werden kann. Dies wird durch zwei Schlüsselwörter gesteuert:

- `var`, kurz für *Variable*, was bedeutet, dass Sie seinen Inhalt neu zuweisen können.
- `val`, kurz für *Wert*, was bedeutet, dass Sie ihn nur initialisieren können; Sie können ihn nicht neu zuweisen.

Sie definieren ein `var` so:

```
var identifier = initialization
```

Das Schlüsselwort `var` wird gefolgt von dem Bezeichner, einem Gleichheitszeichen und dann dem Initialisierungswert. Der Bezeichner beginnt mit einem Buchstaben oder einem Unterstrich, gefolgt von Buchstaben, Zahlen und Unterstrichen. Groß- und Kleinschreibung werden unterschieden (also sind `thisvalue` und `thisValue` unterschiedlich).

Hier sind einige `var` Definitionen:

```
// VarAndVal/Vars.kt

fun main() {
  var whole = 11              // [1]
  var fractional = 1.4        // [2]
  var words = "Twas Brillig"  // [3]
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

In diesem Buch versehen wir Zeilen mit kommentierten Nummern in eckigen Klammern, damit wir im Text auf sie verweisen können, wie folgt:

- **[1]** Erstellen Sie eine `var` namens `whole` und speichern Sie `11` darin.
- **[2]** Speichern Sie die "Bruchzahl" `1.4` in der `var fractional`.
- **[3]** Speichern Sie etwas Text (einen `String`) in der `var words`.

Beachten Sie, dass `println()` jeden einzelnen Wert als Argument annehmen kann.

Wie der Name *Variable* impliziert, kann eine `var` variieren. Das heißt, Sie können die in einer `var` gespeicherten Daten ändern. Wir sagen, dass eine `var` *veränderlich* ist:

```
// VarAndVal/AVarIsMutable.kt

fun main() {
  var sum = 1
  sum = sum + 2
  sum += 3
  println(sum)
}
/* Output:
6
*/
```

Die Zuweisung `sum = sum + 2` nimmt den aktuellen Wert von `sum`, addiert zwei und weist das Ergebnis zurück in `sum` zu.

Die Zuweisung `sum += 3` bedeutet dasselbe wie `sum = sum + 3`. Der `+=` Operator nimmt den vorher gespeicherten Wert in `sum` und erhöht ihn um `3`, dann weist er dieses neue Ergebnis zurück in `sum` zu.

Den in einer `var` gespeicherten Wert zu ändern, ist eine nützliche Methode, um Änderungen auszudrücken. Wenn jedoch die Komplexität eines Programms zunimmt, ist Ihr Code klarer, sicherer und leichter zu verstehen, wenn die Werte, die durch Ihre Bezeichner dargestellt werden, sich nicht ändern können—das heißt, sie können nicht neu zugewiesen werden. Wir spezifizieren einen unveränderlichen Bezeichner, indem wir statt `var` das Schlüsselwort `val` verwenden. Ein `val` kann nur einmal zugewiesen werden, wenn es erstellt wird:

```
val identifier = initialization
```

Das Schlüsselwort `val` stammt von *Wert* und deutet auf etwas hin, das sich nicht ändern kann—es ist *unveränderlich*. Wählen Sie wann immer möglich `val` anstelle von `var`. Das `Vars.kt`-Beispiel am Anfang dieses Abschnitts kann unter Verwendung von `val`s umgeschrieben werden:

```
// VarAndVal/Vals.kt

fun main() {
  val whole = 11
  // whole = 15 // Error   // [1]
  val fractional = 1.4
  val words = "Twas Brillig"
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

- **[1]** Sobald Sie ein `val` initialisieren, können Sie es nicht neu zuweisen. Wenn wir versuchen, `whole` eine andere Zahl zuzuweisen, meldet sich Kotlin mit der Nachricht “Val kann nicht neu zugewiesen werden.”

Beschreibende Namen für Ihre Bezeichner zu wählen, macht Ihren Code leichter verständlich und reduziert oft die Notwendigkeit für Kommentare. In `Vals.kt` haben Sie keine Ahnung, was `whole` repräsentiert. Wenn Ihr Programm die Zahl 11 speichert, um die Tageszeit darzustellen, zu der Sie Kaffee trinken, ist es offensichtlicher für andere, wenn Sie es `coffeetime` nennen, und leichter zu lesen, wenn es `coffeeTime` ist (gemäß dem Kotlin-Stil, bei dem wir den ersten Buchstaben klein schreiben).

- -

`vars` sind nützlich, wenn sich Daten während der Ausführung des Programms ändern müssen. Dies klingt nach einer häufigen Anforderung, stellt sich jedoch in der Praxis als vermeidbar heraus. Im Allgemeinen sind Ihre Programme leichter zu erweitern und zu pflegen, wenn Sie `vals` verwenden. In seltenen Fällen ist es jedoch zu komplex, ein Problem nur mit `vals` zu lösen. Aus diesem Grund bietet Ihnen Kotlin die Flexibilität von `vars`. Je mehr Zeit Sie jedoch mit `vals` verbringen, desto mehr werden Sie entdecken, dass Sie `vars` fast nie benötigen und dass Ihre Programme ohne sie sicherer und zuverlässiger sind.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Datentypen

> Daten können unterschiedliche *Typen* haben.

Um ein mathematisches Problem zu lösen, schreibt man einen Ausdruck:

```
5.9 + 6
```

Du weißt, dass das Addieren dieser Zahlen eine weitere Zahl ergibt. Kotlin weiß das auch. Du weißt, dass eine davon eine Dezimalzahl (`5.9`) ist, die Kotlin `Double` nennt, und die andere eine ganze Zahl (`6`), die Kotlin `Int` nennt. Du weißt, dass das Ergebnis eine Dezimalzahl ist.

Ein *Typ* (auch *Datentyp* genannt) sagt Kotlin, wie du diese Daten verwenden möchtest. Ein Typ definiert die Menge der Werte, die ein Ausdruck dieses Typs erzeugen kann. Ein Typ definiert auch die Operationen, die auf den Daten durchgeführt werden können, die Bedeutung der Daten und wie Werte dieses Typs gespeichert werden können.

Kotlin verwendet Typen, um zu überprüfen, ob deine Ausdrücke korrekt sind. Im obigen Ausdruck erstellt Kotlin einen neuen Wert des Typs `Double`, um das Ergebnis zu speichern.

Kotlin versucht, sich an deine Bedürfnisse anzupassen. Wenn du es bittest, etwas zu tun, das die Typregeln verletzt, erzeugt es eine Fehlermeldung. Zum Beispiel, versuche, einen `String` und eine Zahl zu addieren:

```
// DataTypes/StringPlusNumber.kt

fun main() {
  println("Sally" + 5.9)
}
/* Output:
Sally5.9
*/
```

Typen sagen Kotlin, wie sie korrekt verwendet werden. In diesem Fall sagen die Typregeln Kotlin, wie man eine Zahl zu einem `String` hinzufügt: indem die beiden Werte angehängt werden und ein `String` erstellt wird, um das Ergebnis zu halten.

Versuchen Sie nun, einen `String` und ein `Double` zu multiplizieren, indem Sie das `+` in `StringPlusNumber.kt` durch ein `*` ersetzen:

```
"Sally" * 5.9
```

Das Kombinieren von Typen auf diese Weise ergibt für Kotlin keinen Sinn, daher gibt es einen Fehler aus.

In `var` & `val` haben wir verschiedene Typen gespeichert. Kotlin hat die Typen für uns ermittelt, basierend darauf, wie wir sie verwendet haben. Dies wird *type inference* genannt.

Wir können ausführlicher sein und den Typ angeben:

```
val identifier: Type = initialization
```

Du beginnst mit dem Schlüsselwort `val` oder `var`, gefolgt vom Bezeichner, einem Doppelpunkt, dem Typ, einem `=`, und dem Initialisierungswert. Anstatt also zu sagen:

```
val n = 1
var p = 1.2
```

Du kannst sagen:

```
val n: Int = 1
var p: Double = 1.2
```

Wir haben Kotlin gesagt, dass `n` ein `Int` und `p` ein `Double` ist, anstatt es den Typ ableiten zu lassen.

Hier sind einige von Kotlins grundlegenden Typen:

```
// DataTypes/Types.kt

fun main() {
  val whole: Int = 11               // [1]
  val fractional: Double = 1.4      // [2]
  val trueOrFalse: Boolean = true   // [3]
  val words: String = "A value"     // [4]
  val character: Char = 'z'         // [5]
  val lines: String = """Triple quotes let
you have many lines
in your string"""                   // [6]
  println(whole)
  println(fractional)
  println(trueOrFalse)
  println(words)
  println(character)
  println(lines)
}
/* Output:
11
1.4
true
A value
z
Triple quotes let
you have many lines
in your string
*/
```

- **[1]** Der `Int`-Datentyp ist ein *Ganzzahltyp*, was bedeutet, dass er nur ganze Zahlen speichert.
- **[2]** Um Bruchzahlen zu speichern, verwenden Sie einen `Double`.
- **[3]** Ein `Boolean`-Datentyp speichert nur die beiden speziellen Werte `true` und `false`.
- **[4]** Ein `String` speichert eine Zeichenfolge. Sie weisen einen Wert mit einem doppelt-umrahmten `String` zu.
- **[5]** Ein `Char` speichert ein Zeichen.
- **[6]** Wenn Sie viele Zeilen und/oder Sonderzeichen haben, umgeben Sie diese mit dreifachen Anführungszeichen (dies ist ein *dreifach-umrahmter `String`*).

Kotlin verwendet Typinferenz, um die Bedeutung gemischter Typen zu bestimmen. Beim Mischen von `Int` und `Double` während der Addition entscheidet Kotlin den Typ für den resultierenden Wert:

```
// DataTypes/Inference.kt

fun main() {
  val n = 1 + 1.2
  println(n)
}
/* Output:
2.2
*/
```

Wenn Sie einen `Int` zu einem `Double` mit Typinferenz hinzufügen, bestimmt Kotlin, dass das Ergebnis `n` ein `Double` ist und stellt sicher, dass es alle Regeln für `Doubles` einhält.

Die Typinferenz von Kotlin ist Teil seiner Strategie, Arbeit für den Programmierer zu übernehmen. Wenn Sie die Typdeklaration weglassen, kann Kotlin sie normalerweise ableiten.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Funktionen

> Eine *Funktion* ist wie ein kleines Programm, das einen eigenen Namen hat und durch das Aufrufen dieses Namens aus einer anderen Funktion ausgeführt (*aufgerufen*) werden kann.

Eine Funktion fasst eine Gruppe von Aktivitäten zusammen und ist die grundlegendste Methode, um Ihre Programme zu organisieren und Code wiederzuverwenden.

Sie übergeben Informationen an eine Funktion, und die Funktion verwendet diese Informationen, um ein Ergebnis zu berechnen und zu erzeugen. Die Grundform einer Funktion ist:

```
fun functionName(p1: Type1, p2: Type2, ...): ReturnType {
  lines of code
  return result
}
```

`p1` und `p2` sind die *Parameter*: die Informationen, die Sie in die Funktion übergeben. Jeder Parameter hat einen Bezeichnernamen (`p1`, `p2`) gefolgt von einem Doppelpunkt und dem Typ dieses Parameters. Die schließende Klammer der Parameterliste wird von einem Doppelpunkt und dem Typ des von der Funktion erzeugten Ergebnisses gefolgt. Die Codezeilen im *Funktionskörper* sind in geschweifte Klammern eingeschlossen. Der Ausdruck nach dem Schlüsselwort `return` ist das Ergebnis, das die Funktion erzeugt, wenn sie abgeschlossen ist.

Ein Parameter definiert, was in eine Funktion übergeben wird — er ist der Platzhalter. Ein Argument ist der tatsächliche Wert, den Sie in die Funktion übergeben.

Die Kombination aus Name, Parametern und Rückgabetyp wird als *Funktionssignatur* bezeichnet.

Hier ist eine einfache Funktion namens `multiplyByTwo()`:

```
// Functions/MultiplyByTwo.kt

fun multiplyByTwo(x: Int): Int {  // [1]
  println("Inside multiplyByTwo") // [2]
  return x * 2
}

fun main() {
  val r = multiplyByTwo(5)        // [3]
  println(r)
}
/* Output:
Inside multiplyByTwo
10
*/
```

- **[1]** Beachten Sie das `fun` Schlüsselwort, den Funktionsnamen und die Parameterliste, die aus einem einzigen Parameter besteht. Diese Funktion nimmt einen `Int` Parameter und gibt einen `Int` zurück.
- **[2]** Diese zwei Zeilen sind der Körper der Funktion. Die letzte Zeile gibt den Wert ihrer Berechnung `x * 2` als Ergebnis der Funktion zurück.
- **[3]** Diese Zeile *ruft* die Funktion mit einem geeigneten Argument auf und erfasst das Ergebnis in `val r`. Ein Funktionsaufruf imitiert die Form seiner Deklaration: den Funktionsnamen, gefolgt von Argumenten in Klammern.

Der Funktionscode wird durch Aufrufen der Funktion ausgeführt, wobei der Funktionsname `multiplyByTwo()` als Abkürzung für diesen Code dient. Aus diesem Grund sind Funktionen die grundlegendste Form der Vereinfachung und Wiederverwendung von Code in der Programmierung. Sie können auch an eine Funktion als Ausdruck mit austauschbaren Werten (den Parametern) denken.

`println()` ist ebenfalls ein Funktionsaufruf – er wird einfach von Kotlin bereitgestellt. Wir beziehen uns auf von Kotlin definierte Funktionen als *Bibliotheksfunktionen.*

Wenn die Funktion kein sinnvolles Ergebnis liefert, ist ihr Rückgabetyp `Unit`. Sie können `Unit` explizit angeben, wenn Sie möchten, aber Kotlin erlaubt es Ihnen, es wegzulassen:

```
// Functions/SayHello.kt

fun sayHello() {
  println("Hallo!")
}

fun sayGoodbye(): Unit {
  println("Auf Wiedersehen!")
}

fun main() {
  sayHello()
  sayGoodbye()
}
/* Output:
Hallo!
Auf Wiedersehen!
*/
```

Sowohl `sayHello()` als auch `sayGoodbye()` geben `Unit` zurück, aber `sayHello()` lässt die explizite Deklaration weg. Die `main()`-Funktion gibt ebenfalls `Unit` zurück.

Wenn eine Funktion nur einen einzigen Ausdruck enthält, können Sie die abgekürzte Syntax mit einem Gleichheitszeichen gefolgt von dem Ausdruck verwenden:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = expression
```

Ein Funktionskörper, der von geschweiften Klammern umgeben ist, wird als *Blockkörper* bezeichnet. Ein Funktionskörper, der die Gleichungssyntax verwendet, wird als *Ausdruckskörper* bezeichnet.

Hier verwendet `multiplyByThree()` einen Ausdruckskörper:

```
// Functions/MultiplyByThree.kt

fun multiplyByThree(x: Int): Int = x * 3

fun main() {
  println(multiplyByThree(5))
}
/* Output:
15
*/
```

Dies ist eine kurze Version, um `return x * 3` innerhalb eines Blockkörpers zu sagen.

Kotlin leitet den Rückgabetyp einer Funktion ab, die einen Ausdruckskörper hat:

```
// Functions/MultiplyByFour.kt

fun multiplyByFour(x: Int) = x * 4

fun main() {
  val result: Int = multiplyByFour(5)
  println(result)
}
/* Output:
20
*/
```

Kotlin leitet ab, dass `multiplyByFour()` ein `Int` zurückgibt.

Kotlin kann Rückgabetypen *nur* für Ausdruckskörper ableiten. Wenn eine Funktion einen Blockkörper hat und Sie ihren Typ weglassen, gibt diese Funktion `Unit` zurück.

- -

Beim Schreiben von Funktionen sollten Sie beschreibende Namen wählen. Dies macht den Code leichter lesbar und kann oft die Notwendigkeit für Codekommentare reduzieren. Wir können nicht immer so beschreibend sein, wie wir es uns wünschen würden, mit den Funktionsnamen in diesem Buch, weil wir durch die Zeilenbreiten eingeschränkt sind.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

`if`-Ausdrücke

> Ein *`if`-Ausdruck* trifft eine Entscheidung.

Das `if`-Schlüsselwort prüft einen Ausdruck, um festzustellen, ob er `wahr` oder `falsch` ist, und führt basierend auf dem Ergebnis eine Aktion aus. Ein Wahr-oder-Falsch-Ausdruck wird als *Boolean* bezeichnet, nach dem Mathematiker George Boole, der die Logik hinter diesen Ausdrücken erfunden hat. Hier ist ein Beispiel mit den Symbolen > (größer als) und < (kleiner als):

```
// IfExpressions/If1.kt

fun main() {
  if (1 > 0)
    println("It's true!")
  if (10 < 11) {
    println("10 < 11")
    println("ten is less than eleven")
  }
}
/* Output:
It's true!
10 < 11
ten is less than eleven
*/
```

Der Ausdruck in den Klammern nach dem `if` muss zu `true` oder `false` ausgewertet werden. Wenn `true`, wird der folgende Ausdruck ausgeführt. Um mehrere Zeilen auszuführen, platziere sie in geschweiften Klammern.

Wir können einen booleschen Ausdruck an einer Stelle erstellen und an einer anderen verwenden:

```
// IfExpressions/If2.kt

fun main() {
  val x: Boolean = 1 >= 1
  if (x)
    println("It's true!")
}
/* Output:
It's true!
*/
```

Da `x` ein `Boolean` ist, kann der `if`-Operator es direkt testen, indem er `if(x)` sagt.

Der `>=`-Operator für Boolean gibt `true` zurück, wenn der Ausdruck auf der linken Seite des Operators *größer oder gleich* dem auf der rechten Seite ist. Ebenso gibt `<=` `true` zurück, wenn der Ausdruck auf der linken Seite *kleiner oder gleich* dem auf der rechten Seite ist.

Das Schlüsselwort `else` ermöglicht es Ihnen, sowohl `true`- als auch `false`-Pfade zu behandeln:

```
// IfExpressions/If3.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else
    println("It's negative or zero")
}
/* Output:
It's negative or zero
*/
```

Das `else`-Schlüsselwort wird nur in Verbindung mit `if` verwendet. Sie sind nicht auf eine einzelne Prüfung beschränkt — Sie können mehrere Kombinationen testen, indem Sie `else` und `if` kombinieren:

```
// IfExpressions/If4.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else if (n == 0)
    println("It's zero")
  else
    println("It's negative")
}
/* Output:
It's negative
*/
```

Hier verwenden wir ==, um zwei Zahlen auf Gleichheit zu prüfen. != testet auf Ungleichheit.

Das typische Muster beginnt mit `if`, gefolgt von so vielen `else if`-Klauseln, wie Sie benötigen, und endet mit einem abschließenden `else` für alles, was nicht zu den vorherigen Tests passt. Wenn ein `if`-Ausdruck eine bestimmte Größe und Komplexität erreicht, verwenden Sie wahrscheinlich stattdessen einen `when`-Ausdruck. `when` wird später im Buch beschrieben, in „when“ Ausdrücke.

Der „Nicht“-Operator `!` testet das Gegenteil eines Booleschen Ausdrucks:

```
// IfExpressions/If5.kt

fun main() {
  val y: Boolean = false
  if (!y)
    println("!y is true")
}
/* Output:
!y is true
*/
```

Um `if(!y)` zu verbalisieren, sagt man “wenn nicht y.”

Das gesamte `if` ist ein Ausdruck, der ein Ergebnis liefern kann:

```
// IfExpressions/If6.kt

fun main() {
  val num = 10
  val result = if (num > 100) 4 else 42
  println(result)
}
/* Output:
42
*/
```

Hier speichern wir den Wert, der durch den gesamten `if` Ausdruck erzeugt wird, in einem Zwischenbezeichner namens `result`. Wenn die Bedingung erfüllt ist, erzeugt der erste Zweig `result`. Wenn nicht, wird der `else` Wert zu `result`.

Lassen Sie uns üben, Funktionen zu erstellen. Hier ist eine, die einen Booleschen Parameter nimmt:

```
// IfExpressions/TrueOrFalse.kt

fun trueOrFalse(exp: Boolean): String {
  if (exp)
    return "It's true!"          // [1]
  return "It's false"            // [2]
}

fun main() {
  val b = 1
  println(trueOrFalse(b < 3))
  println(trueOrFalse(b >= 3))
}
/* Output:
It's true!
It's false
*/
```

Der Boolean-Parameter `exp` wird an die Funktion `trueOrFalse()` übergeben. Wenn das Argument als Ausdruck übergeben wird, wie `b < 3`, wird dieser Ausdruck zuerst ausgewertet und das Ergebnis an die Funktion übergeben. `trueOrFalse()` testet `exp` und wenn das Ergebnis `true` ist, wird Zeile **[1]** ausgeführt, andernfalls wird Zeile **[2]** ausgeführt.

- **[1]** `return` sagt: "Verlasse die Funktion und liefere diesen Wert als Ergebnis der Funktion." Beachten Sie, dass `return` überall in einer Funktion erscheinen kann und nicht am Ende stehen muss.

Anstatt `return` wie im vorherigen Beispiel zu verwenden, können Sie das `else`-Schlüsselwort verwenden, um das Ergebnis als Ausdruck zu erzeugen:

```
// IfExpressions/OneOrTheOther.kt

fun oneOrTheOther(exp: Boolean): String =
  if (exp)
    "True!" // No 'return' necessary
  else
    "False"

fun main() {
  val x = 1
  println(oneOrTheOther(x == 1))
  println(oneOrTheOther(x == 2))
}
/* Output:
True!
False
*/
```

Anstelle von zwei Ausdrücken in `trueOrFalse()` ist `oneOrTheOther()` ein einzelner Ausdruck. Das Ergebnis dieses Ausdrucks ist das Ergebnis der Funktion, sodass der `if`-Ausdruck zum Funktionskörper wird.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

String-Vorlagen

> Eine *`String`-Vorlage* ist eine programmatische Methode, um einen `String` zu erzeugen.

Wenn Sie ein `$` vor einen Bezeichnernamen setzen, fügt die `String`-Vorlage den Inhalt dieses Bezeichners in den `String` ein:

```
// StringTemplates/StringTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")     // [1]
  println("printing a $1")      // [2]
}
/* Output:
Found 42!
printing a $1
*/
```

- **[1]** `$answer` ersetzt den Wert von `answer`.
- **[2]** Wenn dem `$` nichts folgt, das als Programmbezeichner erkennbar ist, passiert nichts Besonderes.

Man kann auch Werte in einen `String` einfügen, indem man die Verkettung (`+`) verwendet:

```
// StringTemplates/StringConcatenation.kt

fun main() {
  val s = "hi\n" // \n is a newline character
  val n = 11
  val d = 3.14
  println("first: " + s + "second: " +
    n + ", third: " + d)
}
/* Output:
first: hi
second: 11, third: 3.14
*/
```

Das Platzieren eines Ausdrucks innerhalb von `${}` wertet ihn aus. Der Rückgabewert wird in einen `String` umgewandelt und in den resultierenden `String` eingefügt:

```
// StringTemplates/ExpressionInTemplate.kt

fun main() {
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [1]
  val x = 11
  println("$x + 4 = ${x + 4}")
}
/* Output:
a
11 + 4 = 15
*/
```

- **[1]** `if(condition) 'a' else 'b'` wird ausgewertet und das Ergebnis wird durch den gesamten `${}` Ausdruck ersetzt.

Wenn eine `Zeichenkette` ein Sonderzeichen enthalten muss, wie zum Beispiel ein Anführungszeichen, können Sie entweder dieses Zeichen mit einem `\` (*Backslash*) entkommen, oder Sie verwenden ein `Zeichenkette` Literal in dreifachen Anführungszeichen:

```
// StringTemplates/TripleQuotes.kt

fun main() {
  val s = "value"
  println("s = \"$s\".")
  println("""s = "$s".""")
}
/* Output:
s = "value".
s = "value".
*/
```

Mit dreifachen Anführungszeichen fügen Sie einen Wert eines Ausdrucks auf die gleiche Weise ein wie bei einem einfach-quotierten `String`.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Zahlentypen

> Verschiedene Zahlentypen werden auf unterschiedliche Weise gespeichert.

Wenn Sie einen Bezeichner erstellen und ihm einen Ganzzahlwert zuweisen, leitet Kotlin den `Int`-Typ ab:

```
// NumberTypes/InferInt.kt

fun main() {
  val million = 1_000_000  // Infers Int
  println(million)
}
/* Output:
1000000
*/
```

Für die Lesbarkeit erlaubt Kotlin Unterstriche innerhalb numerischer Werte.

Die grundlegenden mathematischen Operatoren für Zahlen sind diejenigen, die in den meisten Programmiersprachen verfügbar sind: Addition (+), Subtraktion (-), Division (/), Multiplikation (*) und Modulus (%), was den Rest aus der Ganzzahldivision ergibt:

```
// NumberTypes/Modulus.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator % denominator)
}
/* Output:
9
*/
```

Ganzzahl-Division schneidet das Ergebnis ab:

```
// NumberTypes/IntDivisionTruncates.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator / denominator)
}
/* Output:
1
*/
```

Wenn die Operation das Ergebnis gerundet hätte, wäre die Ausgabe `2`.

Die Reihenfolge der Operationen folgt der grundlegenden Arithmetik:

```
// NumberTypes/OpOrder.kt

fun main() {
  println(45 + 5 * 6)
}
/* Output:
75
*/
```

Die Multiplikation `5 * 6` wird zuerst ausgeführt, gefolgt von der Addition `45 + 30`.

Wenn Sie möchten, dass `45 + 5` zuerst erfolgt, verwenden Sie Klammern:

```
// NumberTypes/OpOrderParens.kt

fun main() {
  println((45 + 5) * 6)
}
/* Output:
300
*/
```

Nun berechnen wir den *Body-Mass-Index* (BMI), der das Gewicht in Kilogramm dividiert durch das Quadrat der Größe in Metern ist. Wenn Sie einen BMI von weniger als `18,5` haben, sind Sie untergewichtig. Zwischen `18,5` und `24,9` ist Normalgewicht. Ein BMI von `25` und höher ist Übergewicht. Dieses Beispiel zeigt auch den bevorzugten Formatierungsstil, wenn Sie die Parameter der Funktion nicht in eine einzige Zeile passen können:

```
// NumberTypes/BMIMetric.kt

fun bmiMetric(
  weight: Double,
  height: Double
): String {
  val bmi = weight / (height * height)  // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}

fun main() {
  val weight = 72.57 // 160 lbs
  val height = 1.727 // 68 inches
  val status = bmiMetric(weight, height)
  println(status)
}
/* Output:
Normal weight
*/
```

- **[1]** Wenn Sie die Klammern entfernen, teilen Sie `weight` durch `height` und multiplizieren dann dieses Ergebnis mit `height`. Das ergibt eine viel größere Zahl und ist die falsche Antwort.

`bmiMetric()` verwendet `Doubles` für das Gewicht und die Größe. Ein `Double` kann sehr große und sehr kleine Gleitkommazahlen aufnehmen.

Hier ist eine Version, die englische Einheiten verwendet, dargestellt durch `Int`-Parameter:

```
// NumberTypes/BMIEnglish.kt

fun bmiEnglish(
  weight: Int,
  height: Int
): String {
  val bmi =
    weight / (height * height) * 703.07 // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}

fun main() {
  val weight = 160
  val height = 68
  val status = bmiEnglish(weight, height)
  println(status)
}
/* Output:
Underweight
*/
```

Warum unterscheidet sich das Ergebnis von `bmiMetric()`, das `Doubles` verwendet? Wenn Sie eine Ganzzahl durch eine andere Ganzzahl teilen, erzeugt Kotlin ein Ganzzahlergebnis. Die Standardmethode, um mit dem Rest während der ganzzahligen Division umzugehen, ist das *Abschneiden*, was bedeutet, "abschneiden und wegwerfen" (es gibt kein Runden). Wenn Sie also `5` durch `2` teilen, erhalten Sie `2`, und `7/10` ist null. Wenn Kotlin `bmi` in Ausdruck **[1]** berechnet, teilt es `160` durch `68 * 68` und erhält null. Es multipliziert dann null mit `703.07`, um null zu erhalten.

Um dieses Problem zu vermeiden, verschieben Sie `703.07` an den Anfang der Berechnung. Die Berechnungen werden dann gezwungen, `Double` zu sein:

```
val bmi = 703.07 * weight / (height * height)
```

Die `Double` Parameter in `bmiMetric()` verhindern dieses Problem. Rechnen Sie so früh wie möglich auf den gewünschten Typ um, um die Genauigkeit zu erhalten.

Alle Programmiersprachen haben Grenzen, was sie innerhalb einer Ganzzahl speichern können. Der `Int` Typ in Kotlin kann Werte zwischen -2^{31} und $+2^{31}-1$ annehmen,

eine Einschränkung der 32-Bit Darstellung von `Int`. Wenn Sie zwei `Int`s addieren oder multiplizieren, die groß genug sind, wird das Ergebnis überlaufen:

```
// NumberTypes/IntegerOverflow.kt

fun main() {
  val i: Int = Int.MAX_VALUE
  println(i + i)
}
/* Output:
-2
*/
```

`Int.MAX_VALUE` ist ein vordefinierter Wert, der die größte Zahl darstellt, die ein `Int` halten kann.

Der Überlauf erzeugt ein Ergebnis, das eindeutig falsch ist, da es sowohl negativ als auch viel kleiner ist, als wir erwarten. Kotlin gibt eine Warnung aus, wann immer es einen potenziellen Überlauf erkennt.

Es liegt in Ihrer Verantwortung als Entwickler, Überläufe zu verhindern. Kotlin kann nicht immer Überläufe während der Kompilierung erkennen, und es verhindert keine Überläufe, da dies eine untragbare Leistungseinbuße zur Folge hätte.

Wenn Ihr Programm große Zahlen enthält, können Sie `Long`s verwenden, die Werte von -2^{63} bis $+2^{63}-1$ aufnehmen. Um ein `val` vom Typ `Long` zu definieren, können Sie den Typ explizit angeben oder ein `L` am Ende eines numerischen Literals hinzufügen, was Kotlin anweist, diesen Wert als `Long` zu behandeln:

```
// NumberTypes/LongConstants.kt

fun main() {
  val i = 0           // Infers Int
  val l1 = 0L         // L creates Long
  val l2: Long = 0    // Explicit type
  println("$l1 $l2")
}
/* Output:
0 0
*/
```

Durch die Verwendung von `Long`s verhindern wir den Überlauf in `IntegerOverflow.kt`:

```
// NumberTypes/UsingLongs.kt

fun main() {
  val i = Int.MAX_VALUE
  println(0L + i + i)              // [1]
  println(1_000_000 * 1_000_000L)  // [2]
}
/* Output:
4294967294
1000000000000
*/
```

Die Verwendung eines numerischen Literals in sowohl **[1]** als auch **[2]** erzwingt `Long`-Berechnungen und ergibt ebenfalls ein Ergebnis vom Typ `Long`. Der Ort, an dem das `L` erscheint, ist unwichtig. Wenn einer der Werte `Long` ist, ist der resultierende Ausdruck `Long`.

Obwohl sie viel größere Werte als `Int`s halten können, haben `Long`s immer noch Größenbeschränkungen:

```
// NumberTypes/BiggestLong.kt

fun main() {
  println(Long.MAX_VALUE)
}
/* Output:
9223372036854775807
*/
```

`Long.MAX_VALUE` ist der größte Wert, den ein `Long` halten kann.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Boolesche Werte

> `if`-Ausdrücke demonstrierten den Operator "nicht" `!`, der einen `Boolescher` Wert negiert. Dieses Kapitel führt mehr in die *Boolesche Algebra* ein.

Wir beginnen mit den Operatoren "und" und "oder":

- `&&` (und): Erzeugt `wahr` nur, wenn der `Boolescher` Ausdruck links vom Operator und der rechts beide `wahr` sind.
- `||` (oder): Erzeugt `wahr`, wenn entweder der Ausdruck links oder rechts vom Operator `wahr` ist, oder wenn beide `wahr` sind.

In diesem Beispiel bestimmen wir, ob ein Geschäft geöffnet oder geschlossen ist, basierend auf der `Stunde`:

```
// Booleans/Open1.kt

fun isOpen1(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status =
    if (hour >= open && hour < closed) // [1]
      true
    else
      false
  println("Open: $status")
}

fun main() = isOpen1(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

`main()` ist ein einzelner Funktionsaufruf, daher können wir einen Ausdruckskörper verwenden, wie in Funktionen beschrieben.

Der `if`-Ausdruck in **[1]** prüft, ob `hour` zwischen der Öffnungszeit und der Schließzeit liegt, daher kombinieren wir die Ausdrücke mit dem `Boolean` `&&` (und).

Der `if`-Ausdruck kann vereinfacht werden. Das Ergebnis des Ausdrucks `if(cond) true else false` ist einfach `cond`:

```
// Booleans/Open2.kt

fun isOpen2(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour >= open && hour < closed
  println("Open: $status")
}

fun main() = isOpen2(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

Lassen Sie uns die Logik umkehren und überprüfen, ob das Geschäft derzeit geschlossen ist. Der "or"-Operator `||` liefert `true`, wenn mindestens eine der Bedingungen erfüllt ist:

```
// Booleans/Closed.kt

fun isClosed(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour < open || hour >= closed
  println("Closed: $status")
}

fun main() = isClosed(6)
/* Output:
Operating hours: 9 - 20
```

```
Closed: true
*/
```

`Boolean`-Operatoren ermöglichen komplizierte Logik in kompakten Ausdrücken. Allerdings kann es leicht verwirrend werden. Streben Sie nach Lesbarkeit und spezifizieren Sie Ihre Absichten explizit.

Hier ist ein Beispiel für einen komplizierten `Boolean`-Ausdruck, bei dem unterschiedliche Auswertungsreihenfolgen zu unterschiedlichen Ergebnissen führen:

```
// Booleans/EvaluationOrder.kt

fun main() {
  val sunny = true
  val hoursSleep = 6
  val exercise = false
  val temp = 55

  // [1]:
  val happy1 = sunny && temp > 50 ||
    exercise && hoursSleep > 7
  println(happy1)

  // [2]:
  val sameHappy1 = (sunny && temp > 50) ||
    (exercise && hoursSleep > 7)
  println(sameHappy1)

  // [3]:
  val notSame =
    (sunny && temp > 50 || exercise) &&
      hoursSleep > 7
  println(notSame)
}
/* Output:
true
true
false
*/
```

Die `Boolean`-Ausdrücke sind `sunny`, `temp > 50`, `exercise` und `hoursSleep > 7`. Wir lesen `happy1` als "Es ist sonnig *und* die Temperatur ist größer als `50` *oder* ich

habe Sport getrieben *und* mehr als 7 Stunden geschlafen." Aber hat && Vorrang vor || oder umgekehrt?

Der Ausdruck in **[1]** verwendet die Standardauswertungsreihenfolge von Kotlin. Dies ergibt dasselbe Ergebnis wie der Ausdruck in **[2]**, da ohne Klammern die "und"-Operationen zuerst ausgewertet werden, dann das "oder". Der Ausdruck in **[3]** verwendet Klammern, um ein anderes Ergebnis zu erzielen. In **[3]** sind wir nur glücklich, wenn wir mindestens 7 Stunden geschlafen haben.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Wiederholung mit `while`

> Computer sind ideal für sich wiederholende Aufgaben.

Die grundlegendste Form der Wiederholung verwendet das Schlüsselwort `while`. Dies wiederholt einen Block, solange der kontrollierende *boolesche Ausdruck* `true` ist:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

Der boolesche Ausdruck wird einmal zu Beginn der Schleife und erneut vor jeder weiteren Iteration durch den Block ausgewertet.

```
// RepetitionWithWhile/WhileLoop.kt

fun condition(i: Int) = i < 100  // [1]

fun main() {
  var i = 0
  while (condition(i)) {         // [2]
    print(".")
    i += 10                      // [3]
  }
}
/* Output:
..........
*/
```

- **[1]** Der Vergleichsoperator `<` liefert ein `Boolescher Wert` Ergebnis, daher leitet Kotlin `Boolescher Wert` als Ergebnistyp für `condition()` ab.
- **[2]** Der bedingte Ausdruck für das `while` besagt: "Wiederhole die Anweisungen im Körper, solange `condition()` `true` zurückgibt."
- **[3]** Der `+=` Operator addiert `10` zu `i` und weist das Ergebnis `i` in einem einzigen Vorgang zu (`i` muss eine `var` sein, damit dies funktioniert). Dies entspricht:

```
i = i + 10
```

Es gibt eine zweite Möglichkeit, while in Verbindung mit dem Schlüsselwort do zu verwenden:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

Das Umschreiben von WhileLoop.kt, um eine do-while-Schleife zu verwenden, ergibt:

```
// RepetitionWithWhile/DoWhileLoop.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (condition(i))
}
/* Output:
..........
*/
```

Der einzige Unterschied zwischen while und do-while besteht darin, dass der Körper von do-while immer mindestens einmal ausgeführt wird, selbst wenn der boolesche Ausdruck anfänglich false ergibt. Bei einem while wird der Körper nie ausgeführt, wenn die Bedingung beim ersten Mal false ist. In der Praxis ist do-while weniger verbreitet als while.

Die Kurzformen der Zuweisungsoperatoren sind für alle arithmetischen Operationen verfügbar: +=, -=, *=, /=, und %=. Hier werden -= und %= verwendet:

```
// RepetitionWithWhile/AssignmentOperators.kt

fun main() {
  var n = 10
  val d = 3
  print(n)
  while (n > d) {
    n -= d
    print(" - $d")
  }
  println(" = $n")

  var m = 10
  print(m)
  m %= d
  println(" % $d = $m")
}
/* Output:
10 - 3 - 3 - 3 = 1
10 % 3 = 1
*/
```

Um den Rest der ganzzahligen Division von zwei natürlichen Zahlen zu berechnen, beginnen wir mit einer `while`-Schleife und verwenden dann den Restoperator.

Das Hinzufügen und Subtrahieren von `1` zu einer Zahl ist so häufig, dass sie eigene Inkrement- und Dekrementoperatoren haben: `++` und `--`. Sie können `i += 1` durch `i++` ersetzen:

```
// RepetitionWithWhile/IncrementOperator.kt

fun main() {
  var i = 0
  while (i < 4) {
    print(".")
    i++
  }
}
/* Output:
....
*/
```

In der Praxis werden `while`-Schleifen nicht zum Iterieren über einen Bereich von Zahlen verwendet. Stattdessen wird die `for`-Schleife verwendet. Dies wird im nächsten Atom behandelt.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Schleifen & Bereiche

> Das Schlüsselwort `for` führt einen Codeblock für jeden Wert in einer Sequenz aus.

Die Menge der Werte kann ein Bereich von Ganzzahlen, ein `String` oder, wie Sie später im Buch sehen werden, eine Sammlung von Elementen sein. Das Schlüsselwort `in` zeigt an, dass Sie durch die Werte schreiten:

```
for (v in values) {
  // Do something with v
}
```

Jedes Mal, wenn die Schleife durchlaufen wird, erhält `v` das nächste Element in `values`.

Hier ist eine `for`-Schleife, die eine Aktion eine feste Anzahl von Malen wiederholt:

```
// LoopingAndRanges/RepeatThreeTimes.kt

fun main() {
  for (i in 1..3) {
    println("Hey $i!")
  }
}
/* Output:
Hey 1!
Hey 2!
Hey 3!
*/
```

Die Ausgabe zeigt, dass der `index i` jeden Wert im Bereich von 1 bis 3 erhält.

Ein *range* ist ein Intervall von Werten, das durch ein Paar von Endpunkten definiert wird. Es gibt zwei grundlegende Arten, ranges zu definieren:

```
// LoopingAndRanges/DefiningRanges.kt

fun main() {
  val range1 = 1..10          // [1]
  val range2 = 0 until 10     // [2]
  println(range1)
  println(range2)
}
/* Output:
1..10
0..9
*/
```

- **[1]** Die Verwendung der `..`-Syntax schließt beide Grenzen im resultierenden Bereich ein.
- **[2]** `until` schließt das Ende aus. Die Ausgabe zeigt, dass `10` nicht Teil des Bereichs ist.

Die Anzeige eines Bereichs erzeugt ein lesbares Format.

Dies summiert die Zahlen von 10 bis 100:

```
// LoopingAndRanges/SumUsingRange.kt

fun main() {
  var sum = 0
  for (n in 10..100) {
    sum += n
  }
  println("sum = $sum")
}
/* Output:
sum = 5005
*/
```

Sie können über einen Bereich in umgekehrter Reihenfolge iterieren. Sie können auch einen `Schritt`-Wert verwenden, um das Intervall vom Standardwert `1` zu ändern:

```
// LoopingAndRanges/ForWithRanges.kt

fun showRange(r: IntProgression) {
  for (i in r) {
    print("$i ")
  }
  print("    // $r")
  println()
}

fun main() {
  showRange(1..5)
  showRange(0 until 5)
  showRange(5 downTo 1)          // [1]
  showRange(0..9 step 2)         // [2]
  showRange(0 until 10 step 3)   // [3]
  showRange(9 downTo 2 step 3)
}
/* Output:
1 2 3 4 5     // 1..5
0 1 2 3 4     // 0..4
5 4 3 2 1     // 5 downTo 1 step 1
0 2 4 6 8     // 0..8 step 2
0 3 6 9     // 0..9 step 3
9 6 3     // 9 downTo 3 step 3
*/
```

- **[1]** `downTo` erzeugt einen absteigenden Bereich.
- **[2]** `step` ändert das Intervall. Hier wird der Bereich in Schritten von zwei statt einem durchlaufen.
- **[3]** `until` kann auch mit `step` verwendet werden. Beachten Sie, wie sich dies auf die Ausgabe auswirkt.

In jedem Fall bilden die Zahlenfolgen eine arithmetische Folge. `showRange()` akzeptiert einen `IntProgression`-Parameter, der ein eingebauter Typ ist, der `Int`-Bereiche beinhaltet. Beachten Sie, dass die `String`-Darstellung jeder `IntProgression`, wie sie im Ausgabekommentar für jede Zeile erscheint, oft anders ist als der Bereich, der in `showRange()` übergeben wird—der `IntProgression` übersetzt die Eingabe in eine gleichwertige gemeinsame Form.

Sie können auch einen Bereich von Zeichen erzeugen. Diese `for`-Schleife iteriert von `a` bis `z`:

```
// LoopingAndRanges/ForWithCharRange.kt

fun main() {
  for (c in 'a'..'z') {
    print(c)
  }
}
/* Output:
abcdefghijklmnopqrstuvwxyz
*/
```

Sie können über einen Bereich von Elementen iterieren, die ganze Mengen sind, wie Ganzzahlen und Zeichen, aber nicht Gleitkommawerte.

Eckige Klammern greifen über den Index auf Zeichen zu. Da wir mit dem Zählen der Zeichen in einem `String` bei Null beginnen, wählt `s[0]` das erste Zeichen des `String s` aus. Die Auswahl von `s.lastIndex` ergibt die letzte Indexnummer:

```
// LoopingAndRanges/IndexIntoString.kt

fun main() {
  val s = "abc"
  for (i in 0..s.lastIndex) {
    print(s[i] + 1)
  }
}
/* Output:
bcd
*/
```

Manchmal beschreiben Leute `s[0]` als "das nullte Zeichen."

Zeichen werden als Zahlen gespeichert, die ihren Unicode[17]-Werten entsprechen. Daher ergibt das Hinzufügen einer Ganzzahl zu einem Zeichen ein neues Zeichen, das dem neuen Codewert entspricht:

[17] https://en.wikipedia.org/wiki/Unicode

```
// LoopingAndRanges/AddingIntToChar.kt

fun main() {
  val ch: Char = 'a'
  println(ch + 25)
  println(ch < 'z')
}
/* Output:
z
true
*/
```

Die zweite `println()` zeigt, dass man Zeichencodes vergleichen kann.

Eine `for`-Schleife kann direkt über `String`s iterieren:

```
// LoopingAndRanges/IterateOverString.kt

fun main() {
  for (ch in "Jnskhm ") {
    print(ch + 1)
  }
}
/* Output:
Kotlin!
*/
```

`ch` empfängt nacheinander jedes Zeichen.

Im folgenden Beispiel durchläuft die Funktion `hasChar()` den `String` `s` und prüft, ob er ein bestimmtes Zeichen `ch` enthält. Das `return` in der Mitte der Funktion stoppt die Funktion, sobald die Antwort gefunden wird:

```
// LoopingAndRanges/HasChar.kt

fun hasChar(s: String, ch: Char): Boolean {
  for (c in s) {
    if (c == ch) return true
  }
  return false
}

fun main() {
  println(hasChar("kotlin", 't'))
  println(hasChar("kotlin", 'a'))
}
/* Output:
true
false
*/
```

Das nächste Atom zeigt, dass `hasChar()` unnötig ist — Sie können stattdessen die eingebaute Syntax verwenden.

Wenn Sie einfach eine Aktion eine feste Anzahl von Malen wiederholen möchten, können Sie `repeat()` anstelle einer `for`-Schleife verwenden:

```
// LoopingAndRanges/RepeatHi.kt

fun main() {
  repeat(2) {
    println("hi!")
  }
}
/* Output:
hi!
hi!
*/
```

`repeat()` ist eine Standardbibliotheksfunktion, kein Schlüsselwort. Sie werden viel später im Buch sehen, wie sie erstellt wurde.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Das `in` Schlüsselwort

Das `in` Schlüsselwort prüft, ob ein Wert innerhalb eines Bereichs liegt.

```
// InKeyword/MembershipInRange.kt

fun main() {
  val percent = 35
  println(percent in 1..100)
}
/* Output:
true
*/
```

In Booleans haben Sie gelernt, Grenzen explizit zu überprüfen:

```
// InKeyword/MembershipUsingBounds.kt

fun main() {
  val percent = 35
  println(0 <= percent && percent <= 100)
}
/* Output:
true
*/
```

`0 <= x && x <= 100` ist logisch gleichwertig zu `x in 0..100`. IntelliJ IDEA schlägt vor, die erste Form automatisch durch die zweite zu ersetzen, da diese einfacher zu lesen und zu verstehen ist.

Das Schlüsselwort `in` wird sowohl für Iteration als auch für Mitgliedschaft verwendet. Ein `in` innerhalb des Steuerungsausdrucks einer `for`-Schleife bedeutet Iteration, andernfalls prüft `in` die Mitgliedschaft:

```
// InKeyword/IterationVsMembership.kt

fun main() {
  val values = 1..3
  for (v in values) {
    println("iteration $v")
  }
  val v = 2
  if (v in values)
    println("$v is a member of $values")
}
/* Output:
iteration 1
iteration 2
iteration 3
2 is a member of 1..3
*/
```

Das `in` Schlüsselwort ist nicht nur auf Bereiche beschränkt. Sie können auch überprüfen, ob ein Zeichen Teil eines `String` ist. Das folgende Beispiel verwendet `in` anstelle von `hasChar()` aus dem vorherigen Atom:

```
// InKeyword/InString.kt

fun main() {
  println('t' in "kotlin")
  println('a' in "kotlin")
}
/* Output:
true
false
*/
```

Später im Buch wirst du sehen, dass `in` auch mit anderen Typen funktioniert.

Hier prüft `in`, ob ein Zeichen zu einem Bereich von Zeichen gehört:

```
// InKeyword/CharRange.kt

fun isDigit(ch: Char) = ch in '0'..'9'

fun notDigit(ch: Char) =
  ch !in '0'..'9'               // [1]

fun main() {
  println(isDigit('a'))
  println(isDigit('5'))
  println(notDigit('z'))
}
/* Output:
false
true
true
*/
```

- **[1]** `!in` prüft, dass ein Wert nicht zu einem Bereich gehört.

Sie können einen `Double`-Bereich erstellen, aber Sie können ihn nur verwenden, um die Zugehörigkeit zu überprüfen:

```
// InKeyword/FloatingPointRange.kt

fun inFloatRange(n: Double) {
  val r = 1.0..10.0
  println("$n in $r? ${n in r}")
}

fun main() {
  inFloatRange(0.999999)
  inFloatRange(5.0)
  inFloatRange(10.0)
  inFloatRange(10.0000001)
}
/* Output:
0.999999 in 1.0..10.0? false
5.0 in 1.0..10.0? true
10.0 in 1.0..10.0? true
10.0000001 in 1.0..10.0? false
*/
```

Sie können nur . . verwenden, um einen Gleitkomma-Bereich in Kotlin zu definieren.

Sie können überprüfen, ob ein `String` ein Mitglied eines Bereichs von `Strings` ist:

```
// InKeyword/StringRange.kt

fun main() {
  println("ab" in "aa".."az")
  println("ba" in "aa".."az")
}
/* Output:
true
false
*/
```

Hier verwendet Kotlin den alphabetischen Vergleich.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Ausdrücke & Anweisungen

> *Anweisungen* und *Ausdrücke* sind die kleinsten nützlichen Codefragmente in den meisten Programmiersprachen.

Es gibt einen grundlegenden Unterschied: Eine Anweisung hat eine Wirkung, erzeugt jedoch kein Ergebnis. Ein Ausdruck erzeugt immer ein Ergebnis.

Da eine Anweisung kein Ergebnis erzeugt, muss sie den Zustand ihrer Umgebung ändern, um nützlich zu sein. Anders gesagt: "Eine Anweisung wird wegen ihrer *Seiteneffekte* aufgerufen" (das heißt, was sie *andere* als ein Ergebnis zu erzeugen tut). Als Merkhilfe:

> *Eine Anweisung ändert den Zustand.*

Eine Definition von "ausdrücken" ist "herauspressen", wie in "den Saft aus einer Orange ausdrücken". Also

> *Ein Ausdruck drückt aus.*

Das heißt, er erzeugt ein Ergebnis.

Die `for`-Schleife ist eine Anweisung in Kotlin. Sie kann nicht zugewiesen werden, da es kein Ergebnis gibt:

```
// ExpressionsStatements/ForIsAStatement.kt

fun main() {
  // Can't do this:
  // val f = for(i in 1..10) {}
  // Compiler error message:
  // for is not an expression, and
  // only expressions are allowed here
}
```

Eine `for-Schleife` wird aufgrund ihrer Nebeneffekte verwendet.

Ein Ausdruck erzeugt einen Wert, der zugewiesen oder als Teil eines anderen Ausdrucks verwendet werden kann, während eine Anweisung immer ein Top-Level-Element ist.

Jeder Funktionsaufruf ist ein Ausdruck. Selbst wenn die Funktion `Unit` zurückgibt und nur aufgrund ihrer Nebeneffekte aufgerufen wird, kann das Ergebnis dennoch zugewiesen werden:

```
// ExpressionsStatements/UnitReturnType.kt

fun unitFun() = Unit

fun main() {
  println(unitFun())
  val u1: Unit = println(42)
  println(u1)
  val u2 = println(0) // Type inference
  println(u2)
}
/* Output:
kotlin.Unit
42
kotlin.Unit
0
kotlin.Unit
*/
```

Der `Unit`-Typ enthält einen einzelnen Wert namens `Unit`, den Sie direkt zurückgeben können, wie in `unitFun()` zu sehen ist. Der Aufruf von `println()` gibt ebenfalls

`Unit` zurück. Das `val u1` erfasst den Rückgabewert von `println()` und ist explizit als `Unit` deklariert, während `u2` Typinferenz verwendet.

`if` erzeugt einen Ausdruck, sodass Sie dessen Ergebnis zuweisen können:

```
// ExpressionsStatements/AssigningAnIf.kt

fun main() {
  val result1 = if (11 > 42) 9 else 5

  val result2 = if (1 < 2) {
    val a = 11
    a + 42
  } else 42

  val result3 =
    if ('x' < 'y')
      println("x < y")
    else
      println("x > y")

  println(result1)
  println(result2)
  println(result3)
}
/* Output:
x < y
5
53
kotlin.Unit
*/
```

Die erste Ausgabelinie ist `x < y`, obwohl `result3` erst am Ende von `main()` angezeigt wird. Dies geschieht, weil die Auswertung von `result3` `println()` aufruft und die Auswertung erfolgt, wenn `result3` definiert wird.

Beachten Sie, dass `a` innerhalb des Codeblocks für `result2` definiert ist. Das Ergebnis des letzten Ausdrucks wird zum Ergebnis des `if`-Ausdrucks; hier ist es die Summe von 11 und 42. Aber was ist mit `a`? Sobald Sie den Codeblock verlassen (außerhalb der geschweiften Klammern gehen), können Sie nicht mehr auf `a` zugreifen. Es ist *vorübergehend* und wird verworfen, sobald Sie den *Gültigkeitsbereich* dieses Blocks verlassen.

Der Inkrementoperator `i++` ist auch ein Ausdruck, selbst wenn er wie eine Anweisung aussieht. Kotlin folgt dem Ansatz von C-ähnlichen Sprachen und bietet zwei Versionen von Inkrement- und Dekrementoperatoren mit leicht unterschiedlichen Semantiken. Der Präfix-Operator erscheint vor dem Operanden, wie in `++i`, und gibt den Wert zurück, nachdem das Inkrement erfolgt ist. Sie können es lesen als "zuerst das Inkrement durchführen, dann den resultierenden Wert zurückgeben". Der Postfix-Operator wird nach dem Operanden platziert, wie in `i++`, und gibt den Wert von `i` zurück, bevor das Inkrement erfolgt. Sie können es lesen als "zuerst das Ergebnis erzeugen, dann das Inkrement durchführen".

```
// ExpressionsStatements/PostfixVsPrefix.kt

fun main() {
  var i = 10
  println(i++)
  println(i)
  var j = 20
  println(++j)
  println(j)
}
/* Output:
10
11
21
21
*/
```

Der Dekrementoperator hat auch zwei Versionen: `--i` und `i--`. Die Verwendung von Inkrement- und Dekrementoperatoren innerhalb anderer Ausdrücke wird nicht empfohlen, da dies zu verwirrendem Code führen kann:

```
// ExpressionsStatements/Confusing.kt

fun main() {
  var i = 1
  println(i++ + ++i)
}
```

Versuchen Sie zu erraten, was die Ausgabe sein wird, und überprüfen Sie es dann.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Zusammenfassung 1

> Dieses Atom fasst die Atome in Abschnitt I zusammen und überprüft sie, beginnend bei Hallo, Welt! und endend mit Ausdrücke & Anweisungen.

Wenn Sie ein erfahrener Programmierer sind, sollte dies Ihr erstes Atom sein. Neue Programmierer sollten dieses Atom lesen und die Übungen als Überprüfung von Abschnitt I durchführen.

Wenn Ihnen etwas unklar ist, studieren Sie das zugehörige Atom zu diesem Thema (die Unterüberschriften entsprechen den Atomtiteln).

Hallo, Welt

Kotlin unterstützt sowohl `//` Einzelzeilenkommentare als auch `/*`-bis-`*/` Mehrzeilenkommentare. Der Einstiegspunkt eines Programms ist die Funktion `main()`:

```
// Summary1/Hello.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

Die erste Zeile jedes Beispiels in diesem Buch ist ein Kommentar, der den Namen des Unterverzeichnisses des Atoms enthält, gefolgt von einem `/` und dem Dateinamen. Alle extrahierten Code-Beispiele finden Sie unter AtomicKotlin.com[18].

`println()` ist eine Standardbibliotheksfunktion, die einen einzelnen `String`-Parameter (oder einen Parameter, der in einen `String` konvertiert werden kann) nimmt.

[18] http://AtomicKotlin.com

`println()` bewegt den Cursor nach der Ausgabe seines Parameters in eine neue Zeile, während `print()` den Cursor in derselben Zeile lässt.

Kotlin erfordert kein Semikolon am Ende eines Ausdrucks oder einer Anweisung. Semikolons sind nur notwendig, um mehrere Ausdrücke oder Anweisungen in einer einzigen Zeile zu trennen.

`var` & `val`, Datentypen

Um einen unveränderlichen Bezeichner zu erstellen, verwenden Sie das Schlüsselwort `val`, gefolgt vom Bezeichnernamen, einem Doppelpunkt und dem Typ für diesen Wert. Fügen Sie dann ein Gleichheitszeichen und den Wert hinzu, der diesem `val` zugewiesen werden soll:

```
val identifier: Type = initialization
```

Sobald einem `val` ein Wert zugewiesen wurde, kann er nicht neu zugewiesen werden.

Kotlons *Typinferenz* kann normalerweise den Typ automatisch bestimmen, basierend auf dem Initialisierungswert. Dies führt zu einer einfacheren Definition:

```
val identifier = initialization
```

Beide der folgenden sind gültig:

```
val daysInFebruary = 28
val daysInMarch: Int = 31
```

Eine `var` (Variable) Definition sieht gleich aus, indem `var` anstelle von `val` verwendet wird:

```
var identifier1 = initialization
var identifier2: Type = initialization
```

Im Gegensatz zu einem `val` können Sie ein `var` ändern, daher ist Folgendes zulässig:

```
var hoursSpent = 20
hoursSpent = 25
```

Allerdings kann der *Typ* nicht geändert werden, sodass Sie einen Fehler erhalten, wenn Sie sagen:

```
hoursSpent = 30.5
```

Kotlin leitet den `Int`-Typ ab, wenn `hoursSpent` definiert wird, daher wird es die Änderung in einen Gleitkommawert nicht akzeptieren.

Funktionen

Funktionen sind *benannte Unterprogramme*:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType {
  // Lines of code ...
  return result
}
```

Das Schlüsselwort `fun` wird gefolgt vom Funktionsnamen und der Parameterliste in Klammern. Jeder Parameter muss einen expliziten Typ haben, da Kotlin die Parameterarten nicht ableiten kann. Die Funktion selbst hat einen Typ, der auf die gleiche Weise definiert wird wie bei `var` oder `val` (ein Doppelpunkt gefolgt vom Typ). Der Typ der Funktion ist der Typ des zurückgegebenen Ergebnisses.

Die Funktionssignatur wird gefolgt vom Funktionskörper, der in geschweiften Klammern enthalten ist. Die `return`-Anweisung liefert den Rückgabewert der Funktion.

Sie können eine abgekürzte Syntax verwenden, wenn die Funktion aus einem einzelnen Ausdruck besteht:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = result
```

Diese Form wird als *Ausdruckskörper* bezeichnet. Anstelle einer öffnenden geschweiften Klammer verwenden Sie ein Gleichheitszeichen gefolgt vom Ausdruck. Sie können den Rückgabetyp weglassen, weil Kotlin ihn ableitet.

Hier ist eine Funktion, die den Würfel ihres Parameters produziert, und eine andere, die ein Ausrufezeichen zu einem `String` hinzufügt:

```
// Summary1/BasicFunctions.kt

fun cube(x: Int): Int {
  return x * x * x
}

fun bang(s: String) = s + "!"

fun main() {
  println(cube(3))
  println(bang("pop"))
}
/* Output:
27
pop!
*/
```

`cube()` hat einen Blockkörper mit einer expliziten `return`-Anweisung. `bang()` ist ein Ausdruckskörper, der den Rückgabewert der Funktion erzeugt. Kotlin leitet den Rückgabetyp von `bang()` als `String` ab.

Boolesche Werte

Für die Boolesche Algebra bietet Kotlin Operatoren wie:

- `!` (nicht) negiert den Wert logisch (wandelt `true` in `false` und umgekehrt).
- `&&` (und) gibt `true` nur zurück, wenn *beide* Bedingungen `true` sind.
- `||` (oder) gibt `true` zurück, wenn mindestens eine der Bedingungen `true` ist.

```
// Summary1/Booleans.kt

fun main() {
  val opens = 9
  val closes = 20
  println("Operating hours: $opens - $closes")
  val hour = 6
  println("Current time: " + hour)

  val isOpen = hour >= opens && hour < closes
  println("Open: " + isOpen)
  println("Not open: " + !isOpen)

  val isClosed = hour < opens || hour >= closes
  println("Closed: " + isClosed)
}
/* Output:
Operating hours: 9 - 20
Current time: 6
Open: false
Not open: true
Closed: true
*/
```

Der Initialisierer von `isOpen` verwendet `&&`, um zu testen, ob beide Bedingungen `true` sind. Die erste Bedingung `hour >= opens` ist `false`, sodass das Ergebnis des gesamten Ausdrucks `false` wird. Der Initialisierer für `isClosed` verwendet `||`, was `true` ergibt, wenn mindestens eine der Bedingungen `true` ist. Der Ausdruck `hour < opens` ist `true`, daher ist der gesamte Ausdruck `true`.

`if`-Ausdrücke

Da `if` ein Ausdruck ist, liefert es ein Ergebnis. Dieses Ergebnis kann einer `var` oder `val` zugewiesen werden. Hier sehen Sie auch die Verwendung des Schlüsselworts `else`:

```
// Summary1/IfResult.kt

fun main() {
  val result = if (99 < 100) 4 else 42
  println(result)
}
/* Output:
4
*/
```

Entweder Zweig eines `if`-Ausdrucks kann ein mehrzeiliger Codeblock sein, der von geschweiften Klammern umgeben ist:

```
// Summary1/IfExpression.kt

fun main() {
  val activity = "swimming"
  val hour = 10

  val isOpen = if (
    activity == "swimming" ||
    activity == "ice skating") {
    val opens = 9
    val closes = 20
    println("Operating hours: " +
      opens + " - " + closes)
    hour >= opens && hour < closes
  } else {
    false
  }
  println(isOpen)
}
/* Output:
Operating hours: 9 - 20
true
*/
```

Ein Wert, der innerhalb eines Codeblocks definiert ist, wie `opens`, ist außerhalb des Gültigkeitsbereichs dieses Blocks nicht zugänglich. Da sie *global* für den `if`-Ausdruck definiert sind, sind `activity` und `hour` innerhalb des `if`-Ausdrucks zugänglich.

Das Ergebnis eines `if`-Ausdrucks ist das Ergebnis des letzten Ausdrucks des gewählten Zweigs. Hier ist es `hour >= opens && hour <= closes`, was `true` ist.

String-Vorlagen

Sie können einen Wert innerhalb eines `String` mit Hilfe von `String`-Vorlagen einfügen. Verwenden Sie ein `$` vor dem Bezeichnernamen:

```
// Summary1/StrTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")           // [1]
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [2]
  println("printing a $1")            // [3]
}
/* Output:
Found 42!
a
printing a $1
*/
```

- **[1]** `$answer` ersetzt den Wert, der in `answer` enthalten ist.
- **[2]** `${if(condition) 'a' else 'b'}` wertet den Ausdruck innerhalb von `${}` aus und ersetzt das Ergebnis.
- **[3]** Wenn dem `$` etwas folgt, das nicht als Programmkennzeichner erkennbar ist, passiert nichts Besonderes.

Verwenden Sie dreifach-angeführte `Strings`, um mehrzeiligen Text oder Text mit Sonderzeichen zu speichern:

```
// Summary1/ThreeQuotes.kt

fun json(q: String, a: Int) = """{
  "question" : "$q",
  "answer" : $a
}"""

fun main() {
  println(json("The Ultimate", 42))
}
/* Output:
{
  "question" : "The Ultimate",
  "answer" : 42
}
*/
```

Sie müssen keine Sonderzeichen wie " innerhalb eines dreifach-umrahmten `String` maskieren. (In einem regulären `String` schreiben Sie `\"`, um ein Anführungszeichen einzufügen). Wie bei normalen `Strings` können Sie einen Bezeichner oder einen Ausdruck mit `$` innerhalb eines dreifach-umrahmten `String` einfügen.

Zahlentypen

Kotlin bietet Ganzzahltypen (`Int`, `Long`) und Fließkommatypen (`Double`). Eine Ganzzahlenkonstante ist standardmäßig `Int` und `Long`, wenn Sie ein `L` anhängen. Eine Konstante ist `Double`, wenn sie einen Dezimalpunkt enthält:

```
// Summary1/NumberTypes.kt

fun main() {
  val n = 1000    // Int
  val l = 1000L   // Long
  val d = 1000.0  // Double
  println("$n $l $d")
}
/* Output:
1000 1000 1000.0
*/
```

Ein `Int` hält Werte zwischen -2^{31} und $+2^{31}-1$. Ganzzahlenwerte können einen Überlauf verursachen; zum Beispiel verursacht das Hinzufügen von irgendetwas zu `Int.MAX_VALUE` einen Überlauf:

```
// Summary1/Overflow.kt

fun main() {
  println(Int.MAX_VALUE + 1)
  println(Int.MAX_VALUE + 1L)
}
/* Output:
-2147483648
2147483648
*/
```

Im zweiten `println()`-Statement fügen wir `L` zu `1` hinzu, wodurch der gesamte Ausdruck vom Typ `Long` wird, was den Überlauf verhindert. (Ein `Long` kann Werte zwischen -2^{63} und $+2^{63}-1$ aufnehmen).

Wenn Sie ein `Int` durch ein anderes `Int` teilen, erzeugt Kotlin ein `Int`-Ergebnis, und ein verbleibender Rest wird abgeschnitten. Also ergibt `1/2` `0`. Wenn ein `Double` beteiligt ist, wird das `Int` vor der Operation zu `Double` hochgestuft, sodass `1.0/2` `0.5` ergibt.

Man könnte erwarten, dass `d1` im Folgenden `3.4` ergibt:

```
// Summary1/Truncation.kt

fun main() {
  val d1: Double = 3.0 + 2 / 5
  println(d1)
  val d2: Double = 3 + 2.0 / 5
  println(d2)
}
/* Output:
3.0
3.4
*/
```

Aufgrund der Auswertungsreihenfolge tut es das nicht. Kotlin teilt zuerst `2` durch `5`, und Ganzzahlmathematik ergibt `0`, was zu einem Ergebnis von `3.0` führt. Dieselbe

Auswertungsreihenfolge *erzeugt* das erwartete Ergebnis für `d2`. Die Division von `2.0` durch `5` ergibt `0.4`. Die `3` wird zu einem `Double` hochgestuft, weil wir es zu einem `Double` (`0.4`) addieren, was `3.4` ergibt.

Das Verständnis der Auswertungsreihenfolge hilft Ihnen zu entschlüsseln, was ein Programm macht, sowohl mit logischen Operationen (Boolean-Ausdrücken) als auch mit mathematischen Operationen. Wenn Sie sich über die Auswertungsreihenfolge unsicher sind, verwenden Sie Klammern, um Ihre Absicht zu erzwingen. Dies macht es auch für diejenigen, die Ihren Code lesen, klar.

Wiederholung mit `while`

Eine `while`-Schleife läuft weiter, solange der kontrollierende *Boolean-Ausdruck* `true` ergibt:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

Der *Boolean expression* wird einmal zu Beginn der Schleife und erneut vor jeder weiteren Iteration ausgewertet.

```
// Summary1/While.kt

fun testCondition(i: Int) = i < 100

fun main() {
  var i = 0
  while (testCondition(i)) {
    print(".")
    i += 10
  }
}
/* Output:
..........
*/
```

Kotlin leitet `Boolean` als Ergebnistyp für `testCondition()` ab.

Die Kurzformen der Zuweisungsoperatoren sind für alle mathematischen Operationen verfügbar (`+=`, `-=`, `*=`, `/=`, `%=`). Kotlin unterstützt auch die Inkrement- und Dekrementoperatoren `++` und `--`, sowohl in Präfix- als auch in Postfix-Form.

`while` kann mit dem Schlüsselwort `do` verwendet werden:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

Umschreiben von `While.kt`:

```
// Summary1/DoWhile.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (testCondition(i))
}
/* Output:
..........
*/
```

Der einzige Unterschied zwischen `while` und `do-while` besteht darin, dass der Körper von `do-while` immer mindestens einmal ausgeführt wird, selbst wenn der boolesche Ausdruck beim ersten Mal `false` ergibt.

Schleifen & Bereiche

Viele Programmiersprachen greifen auf ein iterierbares Objekt zu, indem sie durch ganze Zahlen gehen. Kotlins `for` erlaubt es Ihnen, Elemente direkt aus iterierbaren Objekten wie Bereichen und `Strings` zu entnehmen. Zum Beispiel wählt diese `for`-Schleife jedes Zeichen in der Zeichenkette `"Kotlin"` aus:

```
// Summary1/StringIteration.kt

fun main() {
  for (c in "Kotlin") {
    print("$c ")
    // c += 1 // error:
    // val cannot be reassigned
  }
}
/* Output:
K o t l i n
*/
```

`c` kann nicht explizit als entweder `var` oder `val` definiert werden—Kotlin macht es automatisch zu einem `val` und leitet seinen Typ als `Char` ab (man kann den Typ explizit angeben, aber in der Praxis wird dies selten getan).

Sie können durch ganze Zahlenwerte mit *Bereichen* iterieren:

```
// Summary1/RangeOfInt.kt

fun main() {
  for (i in 1..10) {
    print("$i ")
  }
}
/* Output:
1 2 3 4 5 6 7 8 9 10
*/
```

Einen Bereich mit `..` zu erstellen, schließt beide Grenzen ein, aber `until` schließt das obere Ende aus: `1 bis 10` ist dasselbe wie `1..9`. Sie können einen Inkrementwert mit `step` angeben: `1..21 Schritt 3`.

Das `in` Schlüsselwort

Dasselbe `in`, das die Iteration in `for` Schleifen ermöglicht, erlaubt es Ihnen auch, die Zugehörigkeit zu einem Bereich zu überprüfen. `!in` gibt `true` zurück, wenn der getestete Wert *nicht* im Bereich liegt:

```
// Summary1/Membership.kt

fun inNumRange(n: Int) = n in 50..100

fun notLowerCase(ch: Char) = ch !in 'a'..'z'

fun main() {
  val i1 = 11
  val i2 = 100
  val c1 = 'K'
  val c2 = 'k'
  println("$i1 ${inNumRange(i1)}")
  println("$i2 ${inNumRange(i2)}")
  println("$c1 ${notLowerCase(c1)}")
  println("$c2 ${notLowerCase(c2)}")
}
/* Output:
11 false
100 true
K true
k false
*/
```

`in` kann auch verwendet werden, um die Zugehörigkeit zu Gleitkomma-Bereichen zu testen, obwohl solche Bereiche nur mit `..` und nicht mit `until` definiert werden können.

Ausdrücke & Statements

Das kleinste nützliche Codefragment in den meisten Programmiersprachen ist entweder ein *Statement* oder ein *Ausdruck*. Diese haben einen grundlegenden Unterschied:

- *Ein Statement ändert den Zustand.*
- *Ein Ausdruck drückt aus.*

Das heißt, ein Ausdruck liefert ein Ergebnis, während ein Statement das nicht tut. Weil es nichts zurückgibt, muss ein Statement den Zustand seiner Umgebung ändern (das heißt, einen *Seiteneffekt* erzeugen), um etwas Nützliches zu tun.

Fast alles in Kotlin ist ein Ausdruck:

```
val hours = 10
val minutesPerHour = 60
val minutes = hours * minutesPerHour
```

In jedem Fall ist alles rechts vom = ein Ausdruck, der ein Ergebnis liefert, das der Bezeichnung links zugewiesen wird.

Funktionen wie `println()` scheinen kein Ergebnis zu erzeugen, aber da sie immer noch Ausdrücke sind, müssen sie *etwas* zurückgeben. Kotlin hat dafür einen speziellen Typ namens `Unit`:

```
// Summary1/UnitReturn.kt

fun main() {
  val result = println("returns Unit")
  println(result)
}
/* Output:
returns Unit
kotlin.Unit
*/
```

Erfahrene Programmierer sollten nach dem Bearbeiten der Übungen für dieses Atom zu Zusammenfassung 2 gehen.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Abschnitt II: Einführung in Objekte

> *Objekte* sind die Grundlage für zahlreiche moderne Sprachen, einschließlich Kotlin.

In einer *objektorientierten* (OO) Programmiersprache entdecken Sie "Nomen" in dem Problem, das Sie lösen, und übersetzen diese Nomen in Objekte. Objekte halten Daten und führen Aktionen aus. Eine objektorientierte Sprache erstellt und verwendet Objekte.

Kotlin ist nicht nur objektorientiert; es ist auch *funktional*. Funktionale Sprachen konzentrieren sich auf die Aktionen, die Sie ausführen ("Verben"). Kotlin ist eine hybride objekt-funktionale Sprache.

- Dieser Abschnitt erklärt die Grundlagen der objektorientierten Programmierung.
- Abschnitt IV: Funktionale Programmierung führt in die funktionale Programmierung ein.
- Abschnitt V: Objektorientierte Programmierung behandelt die objektorientierte Programmierung im Detail.

Objekte überall

> Objekte speichern Daten mithilfe von *Eigenschaften* (`val`s und `var`s) und führen Operationen mit diesen Daten mithilfe von Funktionen durch.

Einige Definitionen:

- *Klasse*: Definiert Eigenschaften und Funktionen für das, was im Wesentlichen ein neuer Datentyp ist. Klassen werden auch als *benutzerdefinierte Typen* bezeichnet.
- *Mitglied*: Entweder eine Eigenschaft oder eine Funktion einer Klasse.
- *Mitgliedsfunktion*: Eine Funktion, die nur mit einer bestimmten Klasse von Objekten arbeitet.
- *Ein Objekt erstellen*: Ein `val` oder `var` einer Klasse erstellen. Auch als *eine Instanz dieser Klasse erstellen* bezeichnet.

Da Klassen *Zustand* und *Verhalten* definieren, können wir sogar Instanzen von eingebauten Typen wie `Double` oder `Boolean` als Objekte bezeichnen.

Betrachten Sie die `IntRange`-Klasse von Kotlin:

```
// ObjectsEverywhere/IntRanges.kt

fun main() {
  val r1 = IntRange(0, 10)
  val r2 = IntRange(5, 7)
  println(r1)
  println(r2)
}
/* Output:
0..10
5..7
*/
```

Wir erstellen zwei Objekte (Instanzen) der *Klasse* `IntRange`. Jedes Objekt hat seinen eigenen Speicherplatz im Speicher. `IntRange` ist eine Klasse, aber ein bestimmter Bereich `r1` von `0` bis `10` ist ein Objekt, das sich von dem Bereich `r2` unterscheidet.

Für ein `IntRange`-Objekt stehen zahlreiche Operationen zur Verfügung. Einige sind einfach, wie `sum()`, und andere erfordern mehr Verständnis, bevor Sie sie verwenden können. Wenn Sie versuchen, eine aufzurufen, die Argumente benötigt, wird die IDE nach diesen Argumenten fragen.

Um mehr über eine bestimmte Mitgliedsfunktion zu erfahren, schlagen Sie in der Kotlin-Dokumentation[19] nach. Beachten Sie das Lupensymbol im oberen rechten Bereich der Seite. Klicken Sie darauf und geben Sie `IntRange` in das Suchfeld ein. Klicken Sie auf `kotlin.ranges > IntRange` aus der resultierenden Suche. Sie sehen die Dokumentation für die `IntRange`-Klasse. Sie können alle Mitgliedsfunktionen—die *Programmierschnittstelle* (API)—der Klasse studieren. Obwohl Sie die meisten davon zu diesem Zeitpunkt nicht verstehen werden, ist es hilfreich, sich daran zu gewöhnen, in der Kotlin-Dokumentation nachzuschlagen.

Ein `IntRange` ist eine Art von Objekt, und ein charakteristisches Merkmal eines Objekts ist, dass Sie Operationen darauf ausführen. Statt "eine Operation ausführen" sagen wir *eine Mitgliedsfunktion aufrufen*. Um eine Mitgliedsfunktion für ein Objekt aufzurufen, beginnen Sie mit dem Objektbezeichner, dann ein Punkt, dann der Name der Operation:

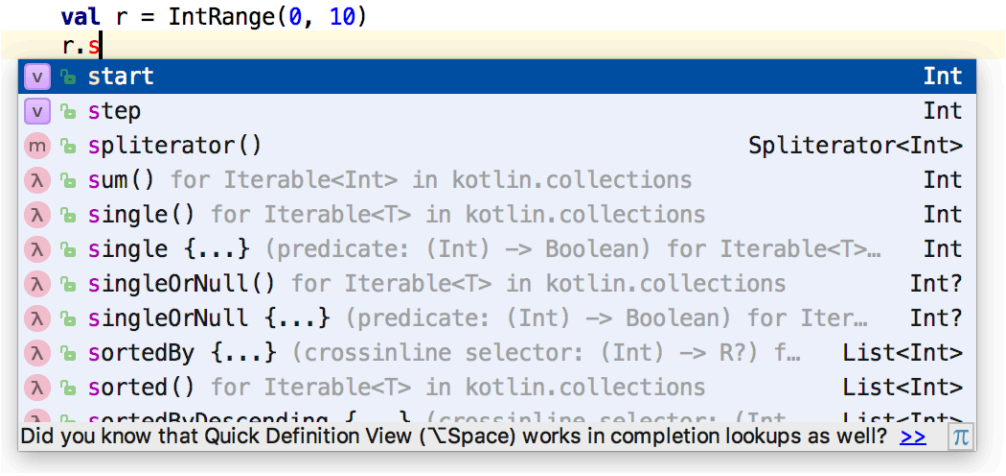
```
// ObjectsEverywhere/RangeSum.kt

fun main() {
  val r = IntRange(0, 10)
  println(r.sum())
}
/* Output:
55
*/
```

Da `sum()` eine Mitgliedsfunktion ist, die für `IntRange` definiert ist, rufen Sie sie auf, indem Sie `r.sum()` schreiben. Dies summiert alle Zahlen in diesem `IntRange`.

Frühere objektorientierte Sprachen verwendeten den Ausdruck "eine Nachricht senden", um das Aufrufen einer Mitgliedsfunktion für ein Objekt zu beschreiben. Manchmal sieht man diese Terminologie noch.

[19] https://kotlinlang.org/api/latest/jvm/stdlib/index.html

Klassen können viele Operationen (Mitgliedsfunktionen) haben. Es ist einfach, Klassen mit einer IDE (integrierte Entwicklungsumgebung) zu erkunden, die eine Funktion namens *Code-Vervollständigung* enthält. Wenn Sie zum Beispiel `.s` nach einem Objektbezeichner in IntelliJ IDEA eingeben, zeigt es alle Mitglieder dieses Objekts an, die mit `s` beginnen:



Code-Vervollständigung

Versuchen Sie, die Code-Vervollständigung bei anderen Objekten zu verwenden. Zum Beispiel können Sie einen `String` umkehren oder alle Zeichen in Kleinbuchstaben umwandeln:

```
// ObjectsEverywhere/Strings.kt

fun main() {
  val s = "AbcD"
  println(s.reversed())
  println(s.lowercase())
}
/* Output:
DcbA
abcd
*/
```

Du kannst einen `String` leicht in einen `integer` umwandeln und zurück:

```
// ObjectsEverywhere/Conversion.kt

fun main() {
  val s = "123"
  println(s.toInt())
  val i = 123
  println(i.toString())
}
/* Output:
123
123
*/
```

Später im Buch besprechen wir Strategien, um Situationen zu bewältigen, wenn der `String`, den Sie konvertieren möchten, keinen korrekten Integer-Wert darstellt.

Sie können auch von einem Zahlentyp zu einem anderen konvertieren. Um Verwirrung zu vermeiden, sind Konvertierungen zwischen Zahlentypen explizit. Zum Beispiel konvertieren Sie ein `Int i` zu einem `Long`, indem Sie `i.toLong()` aufrufen, oder zu einem `Double` mit `i.toDouble()`:

```
// ObjectsEverywhere/NumberConversions.kt

fun fraction(numerator: Long, denom: Long) =
  numerator.toDouble() / denom

fun main() {
  val num = 1
  val den = 2
  val f = fraction(num.toLong(), den.toLong())
  println(f)
}
/* Output:
0.5
*/
```

Gut definierte Klassen sind für einen Programmierer leicht zu verstehen und erzeugen Code, der leicht zu lesen ist.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Klassen erstellen

> Sie können nicht nur vordefinierte Typen wie `IntRange` und `String` verwenden, sondern auch Ihre eigenen Objekttypen erstellen.

Tatsächlich besteht ein Großteil der Aktivitäten in der objektorientierten Programmierung darin, neue Typen zu erstellen. Sie erstellen neue Typen, indem Sie *Klassen* definieren.

Ein Objekt ist ein Teil der Lösung für ein Problem, das Sie zu lösen versuchen. Beginnen Sie damit, Objekte als Ausdruck von Konzepten zu betrachten. Als erste Annäherung, wenn Sie ein "Ding" in Ihrem Problem entdecken, repräsentieren Sie dieses Ding als Objekt in Ihrer Lösung.

Angenommen, Sie möchten ein Programm erstellen, um Tiere in einem Zoo zu verwalten. Es ist sinnvoll, die verschiedenen Tierarten basierend darauf zu kategorisieren, wie sie sich verhalten, welche Bedürfnisse sie haben, mit welchen Tieren sie sich vertragen und mit welchen sie kämpfen. Alles, was eine Tierart unterscheidet, wird in der Klassifizierung des Objekts dieses Tieres erfasst. Kotlin verwendet das Schlüsselwort `class`, um einen neuen Objekttyp zu erstellen:

```
// CreatingClasses/Animals.kt

// Create some classes:
class Giraffe
class Bear
class Hippo

fun main() {
  // Create some objects:
  val g1 = Giraffe()
  val g2 = Giraffe()
  val b = Bear()
  val h = Hippo()

  // Each object() is unique:
```

```
  println(g1)
  println(g2)
  println(h)
  println(b)
}
/* Sample output:
Giraffe@28d93b30
Giraffe@1b6d3586
Hippo@4554617c
Bear@74a14482
*/
```

Um eine Klasse zu definieren, beginnen Sie mit dem Schlüsselwort `class`, gefolgt von einem Bezeichner für Ihre neue Klasse. Der Klassenname muss mit einem Buchstaben (A-Z, Groß- oder Kleinbuchstaben) beginnen, kann jedoch Zahlen und Unterstriche enthalten. Nach Konvention wird der erste Buchstabe eines Klassennamens großgeschrieben, während der erste Buchstabe aller `val`s und `var`s kleingeschrieben wird.

`Animals.kt` beginnt mit der Definition von drei neuen Klassen und erstellt dann vier Objekte (auch *Instanzen* genannt) dieser Klassen.

`Giraffe` ist eine Klasse, aber eine bestimmte fünfjährige männliche Giraffe, die in Botswana lebt, ist ein *Objekt.* Jedes Objekt unterscheidet sich von allen anderen, daher geben wir ihnen Namen wie `g1` und `g2`.

Beachten Sie die etwas kryptische Ausgabe der letzten vier Zeilen. Der Teil vor dem `@` ist der Klassenname und die Zahl nach dem `@` ist die Adresse, an der sich das Objekt im Speicher Ihres Computers befindet. Ja, das ist eine Zahl, auch wenn sie einige Buchstaben enthält - das nennt man "hexadezimale Notation"[20]. Jedes Objekt in Ihrem Programm hat seine eigene eindeutige Adresse.

Die hier definierten Klassen (`Giraffe`, `Bear` und `Hippo`) sind so einfach wie möglich: die gesamte Klassendefinition besteht aus einer einzigen Zeile. Komplexere Klassen verwenden geschweifte Klammern (`{` und `}`), um einen *Klassenkörper* zu erstellen, der die Merkmale und Verhaltensweisen dieser Klasse enthält.

Eine innerhalb einer Klasse definierte Funktion gehört zu dieser Klasse. In Kotlin nennen wir sie *Mitgliedsfunktionen* der Klasse. Einige objektorientierte Programmiersprachen wie Java entscheiden sich dafür, sie *Methoden* zu nennen, ein Begriff,

[20] https://en.wikipedia.org/wiki/Hexadecimal

der aus frühen objektorientierten Sprachen wie Smalltalk stammt. Um die funktionale Natur von Kotlin zu betonen, entschieden sich die Designer, den Begriff *Methode* wegzulassen, da einige Anfänger die Unterscheidung verwirrend fanden. Stattdessen wird in der gesamten Sprache der Begriff *Funktion* verwendet.

Wenn es eindeutig ist, sagen wir einfach "Funktion". Wenn wir die Unterscheidung treffen müssen:

- *Mitgliedsfunktionen* gehören zu einer Klasse.
- *Top-Level-Funktionen* existieren für sich und sind nicht Teil einer Klasse.

Hier gehört `bark()` zur `Dog`-Klasse:

```
// CreatingClasses/Dog.kt

class Dog {
  fun bark() = "yip!"
}

fun main() {
  val dog = Dog()
}
```

In `main()` erstellen wir ein `Dog`-Objekt und weisen es `val dog` zu. Kotlin gibt eine Warnung aus, weil wir `dog` nie verwenden.

Mitgliedsfunktionen werden aufgerufen (*invoked*), indem man den Objektnamen verwendet, gefolgt von einem `.` (Punkt), gefolgt vom Funktionsnamen und der Parameterliste. Hier rufen wir die Funktion `meow()` auf und zeigen das Ergebnis an:

```
// CreatingClasses/Cat.kt

class Cat {
  fun meow() = "mrrrow!"
}

fun main() {
  val cat = Cat()
  // Call 'meow()' for 'cat':
  val m1 = cat.meow()
  println(m1)
}
```

```
/* Output:
mrrrow!
*/
```

Eine Mitgliedsfunktion wirkt auf eine bestimmte Instanz einer Klasse. Wenn Sie `meow()` aufrufen, müssen Sie es mit einem Objekt aufrufen. Während des Aufrufs kann `meow()` auf andere Mitglieder dieses Objekts zugreifen.

Beim Aufrufen einer Mitgliedsfunktion verfolgt Kotlin das betreffende Objekt, indem es leise eine Referenz auf dieses Objekt übergibt. Diese Referenz ist innerhalb der Mitgliedsfunktion mit dem Schlüsselwort `this` verfügbar.

Mitgliedsfunktionen haben einen speziellen Zugriff auf andere Elemente innerhalb einer Klasse, indem sie einfach diese Elemente benennen. Sie können den Zugriff auf diese Elemente auch explizit mit `this` *qualifizieren*. Hier ruft `exercise()` `speak()` mit und ohne Qualifizierung auf:

```
// CreatingClasses/Hamster.kt

class Hamster {
  fun speak() = "Squeak! "
  fun exercise() =
    this.speak() +   // Qualified with 'this'
      speak() +      // Without 'this'
      "Running on wheel"
}

fun main() {
  val hamster = Hamster()
  println(hamster.exercise())
}
/* Output:
Squeak! Squeak! Running on wheel
*/
```

In `exercise()`, rufen wir zuerst `speak()` mit einem expliziten `this` auf und lassen dann die Qualifikation weg.

Manchmal sieht man Code, der ein unnötiges explizites `this` enthält. Solcher Code stammt oft von Programmierern, die eine andere Sprache kennen, in der `this` entweder erforderlich ist oder Teil des Stils ist. Die unnötige Verwendung eines

Features ist verwirrend für den Leser, der Zeit damit verbringt, herauszufinden, warum Sie es tun. Wir empfehlen, die unnötige Verwendung von `this` zu vermeiden.

Außerhalb der Klasse muss man `hamster.exercise()` und `hamster.speak()` sagen.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Eigenschaften

> Eine *Eigenschaft* ist ein `var` oder `val`, das Teil einer Klasse ist.

Das Definieren einer Eigenschaft *beibehält den Zustand* innerhalb einer Klasse. Das Beibehalten des Zustands ist der Hauptmotivationsgrund, eine Klasse zu erstellen, anstatt nur eine oder mehrere eigenständige Funktionen zu schreiben.

Eine `var`-Eigenschaft kann neu zugewiesen werden, während eine `val`-Eigenschaft dies nicht kann. Jedes Objekt erhält seinen eigenen Speicher für Eigenschaften:

```
// Properties/Cup.kt

class Cup {
  var percentFull = 0
}

fun main() {
  val c1 = Cup()
  c1.percentFull = 50
  val c2 = Cup()
  c2.percentFull = 100

  println(c1.percentFull)
  println(c2.percentFull)
}
/* Output:
50
100
*/
```

Das Definieren eines `var` oder `val` innerhalb einer Klasse sieht genauso aus wie das Definieren innerhalb einer Funktion. Allerdings wird das `var` oder `val` *Teil* dieser Klasse, und Sie müssen darauf verweisen, indem Sie das Objekt mit *Punktnotation* angeben, wobei Sie einen Punkt zwischen das Objekt und den Namen der Eigenschaft setzen. Sie können die Punktnotation bei jedem Verweis auf `percentFull` sehen.

Die Eigenschaft `percentFull` repräsentiert den Zustand des entsprechenden `Cup`-Objekts. `c1.percentFull` und `c2.percentFull` enthalten unterschiedliche Werte, was zeigt, dass jedes Objekt über seinen eigenen Speicherplatz verfügt.

Eine Mitgliedsfunktion kann auf eine Eigenschaft innerhalb ihres Objekts verweisen, ohne die Punktnotation zu verwenden (das heißt, ohne sie zu *qualifizieren*):

```
// Properties/Cup2.kt

class Cup2 {
  var percentFull = 0
  val max = 100
  fun add(increase: Int): Int {
    percentFull += increase
    if (percentFull > max)
      percentFull = max
    return percentFull
  }
}

fun main() {
  val cup = Cup2()
  cup.add(50)
  println(cup.percentFull)
  cup.add(70)
  println(cup.percentFull)
}
/* Output:
50
100
*/
```

Die `add()`-Mitgliedsfunktion versucht, `increase` zu `percentFull` hinzuzufügen, stellt jedoch sicher, dass es nicht über 100 % hinausgeht.

Eigenschaften und Mitgliedsfunktionen müssen von außerhalb einer Klasse qualifiziert werden.

Man kann Eigenschaften auf oberster Ebene definieren:

```
// Properties/TopLevelProperty.kt

val constant = 42

var counter = 0

fun inc() {
  counter++
}
```

Die Definition eines `val` auf oberster Ebene ist sicher, da es nicht verändert werden kann. Die Definition einer veränderbaren (`var`) Eigenschaft auf oberster Ebene wird jedoch als *Antimuster* angesehen. Wenn Ihr Programm komplizierter wird, wird es schwieriger, den *gemeinsamen veränderbaren Zustand* korrekt zu verstehen. Wenn jeder in Ihrem Code Zugriff auf den `var Zähler` hat, können Sie nicht garantieren, dass er korrekt verändert wird: Während `inc()` den `Zähler` um eins erhöht, könnte ein anderer Teil des Programms den `Zähler` um zehn verringern, was zu schwer nachvollziehbaren Fehlern führt. Es ist am besten, veränderbaren Zustand innerhalb einer Klasse zu schützen. In Sichtbarkeit einschränken wird gezeigt, wie man ihn wirklich verstecken kann.

Zu sagen, dass `var`s verändert werden können, während `val`s dies nicht können, ist eine Vereinfachung. Als Analogie können Sie ein `Haus` als `val` betrachten und ein `Sofa` im `Haus` als `var`. Sie können das `Sofa` verändern, weil es ein `var` ist. Sie können jedoch das `Haus` nicht neu zuweisen, da es ein `val` ist:

```
// Properties/ChangingAVal.kt

class House {
  var sofa: String = ""
}

fun main() {
  val house = House()
  house.sofa = "Simple sleeper sofa: $89.00"
  println(house.sofa)
  house.sofa = "New leather sofa: $3,099.00"
  println(house.sofa)
  // Cannot reassign the val to a new House:
  // house = House()
```

```
}
/* Output:
Simple sleeper sofa: $89.00
New leather sofa: $3,099.00
*/
```

Obwohl `house` ein `val` ist, kann sein Objekt modifiziert werden, weil `sofa` in `class House` ein `var` ist. Die Definition von `house` als `val` verhindert nur, dass es einem neuen Objekt neu zugewiesen wird.

Wenn wir eine Eigenschaft als `val` definieren, kann sie nicht neu zugewiesen werden:

```
// Properties/AnUnchangingVar.kt

class Sofa {
  val cover: String = "Loveseat cover"
}

fun main() {
  var sofa = Sofa()
  // Not allowed:
  // sofa.cover = "New cover"
  // Reassigning a var:
  sofa = Sofa()
}
```

Auch wenn `sofa` eine `var` ist, kann sein Objekt nicht modifiziert werden, weil `cover` in `class Sofa` ein `val` ist. `sofa` kann jedoch einem neuen Objekt zugewiesen werden.

Wir haben über Bezeichner wie `house` und `sofa` gesprochen, als wären sie Objekte. Tatsächlich sind sie *Referenzen* auf Objekte. Eine Möglichkeit, dies zu sehen, ist zu beobachten, dass zwei Bezeichner auf dasselbe Objekt verweisen können:

```
// Properties/References.kt

class Kitchen {
  var table: String = "Round table"
}

fun main() {
  val kitchen1 = Kitchen()
  val kitchen2 = kitchen1
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
  kitchen1.table = "Square table"
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
}
/* Output:
kitchen1: Round table
kitchen2: Round table
kitchen1: Square table
kitchen2: Square table
*/
```

Wenn `kitchen1 table` verändert, sieht `kitchen2` die Änderung. `kitchen1.table` und `kitchen2.table` zeigen die gleiche Ausgabe.

Denken Sie daran, dass `var` und `val` Referenzen anstelle von Objekten steuern. Ein `var` ermöglicht es Ihnen, eine Referenz auf ein anderes Objekt neu zu binden, während ein `val` dies verhindert.

Veränderlichkeit bedeutet, dass ein Objekt seinen Zustand ändern kann. In den obigen Beispielen definieren `class House` und `class Kitchen` veränderliche Objekte, während `class Sofa` unveränderliche Objekte definiert.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Konstruktoren

> Sie initialisieren ein neues Objekt, indem Sie Informationen an einen *Konstruktor* übergeben.

Jedes Objekt ist eine isolierte Welt. Ein Programm ist eine Sammlung von Objekten, daher löst die korrekte Initialisierung jedes einzelnen Objekts einen großen Teil des Initialisierungsproblems. Kotlin enthält Mechanismen, um eine ordnungsgemäße Initialisierung von Objekten zu gewährleisten.

Ein Konstruktor ist wie eine spezielle Mitgliedsfunktion, die ein neues Objekt initialisiert. Die einfachste Form eines Konstruktors ist eine einzeilige Klassendefinition:

```
// Constructors/Wombat.kt

class Wombat

fun main() {
  val wombat = Wombat()
}
```

In `main()`, das Aufrufen von `Wombat()` erstellt ein `Wombat`-Objekt. Wenn Sie aus einer anderen objektorientierten Sprache kommen, könnten Sie erwarten, hier ein `new`-Schlüsselwort zu sehen, aber `new` wäre in Kotlin überflüssig, also wurde es weggelassen.

Sie übergeben Informationen an einen Konstruktor mit einer Parameterliste, genau wie bei einer Funktion. Hier nimmt der `Alien`-Konstruktor ein einziges Argument entgegen:

```
// Constructors/Arg.kt

class Alien(name: String) {
  val greeting = "Poor $name!"
}

fun main() {
  val alien = Alien("Mr. Meeseeks")
  println(alien.greeting)
  // alien.name // Error     // [1]
}
/* Output:
Poor Mr. Meeseeks!
*/
```

Um ein `Alien`-Objekt zu erstellen, ist ein Argument erforderlich (versuchen Sie es ohne eines). `name` initialisiert die `greeting`-Eigenschaft innerhalb des Konstruktors, ist jedoch außerhalb des Konstruktors nicht zugänglich – versuchen Sie, die Zeile **[1]** zu entkommentieren.

Wenn Sie möchten, dass der Konstruktor-Parameter außerhalb des Klassenkörpers zugänglich ist, definieren Sie ihn als `var` oder `val` in der Parameterliste:

```
// Constructors/VisibleArgs.kt

class MutableNameAlien(var name: String)

class FixedNameAlien(val name: String)

fun main() {
  val alien1 =
    MutableNameAlien("Reverse Giraffe")
  val alien2 =
    FixedNameAlien("Krombopulos Michael")

  alien1.name = "Parasite"
  // Can't do this:
  // alien2.name = "Parasite"
}
```

Diese Klassendefinitionen haben keine expliziten Klassenkörper—die Körper sind implizit.

Wenn `name` als `var` oder `val` definiert wird, wird es zu einer Eigenschaft und ist somit außerhalb des Konstruktors zugänglich. `val`-Konstruktorparameter können nicht geändert werden, während `var`-Konstruktorparameter veränderbar sind.

Ihre Klasse kann zahlreiche Konstruktorparameter haben:

```
// Constructors/MultipleArgs.kt

class AlienSpecies(
  val name: String,
  val eyes: Int,
  val hands: Int,
  val legs: Int
) {
  fun describe() =
    "$name with $eyes eyes, " +
      "$hands hands and $legs legs"
}

fun main() {
  val kevin =
    AlienSpecies("Zigerion", 2, 2, 2)
  val mortyJr =
    AlienSpecies("Gazorpian", 2, 6, 2)
  println(kevin.describe())
  println(mortyJr.describe())
}
/* Output:
Zigerion with 2 eyes, 2 hands and 2 legs
Gazorpian with 2 eyes, 6 hands and 2 legs
*/
```

In Komplexe Konstruktoren, werden Sie sehen, dass Konstruktoren auch komplexe Initialisierungslogik enthalten können.

Wenn ein Objekt verwendet wird, wenn ein `String` erwartet wird, ruft Kotlin die `toString()`-Mitgliedsfunktion des Objekts auf. Wenn Sie keine schreiben, erhalten Sie trotzdem eine Standard-`toString()`:

```
// Constructors/DisplayAlienSpecies.kt

fun main() {
  val krombopulosMichael =
    AlienSpecies("Gromflomite", 2, 2, 2)
  println(krombopulosMichael)
}
/* Sample output:
AlienSpecies@4d7e1886
*/
```

Der Standard-`toString()` ist nicht sehr nützlich—er gibt den Klassennamen und die physikalische Adresse des Objekts aus (dies variiert von einer Programmausführung zur nächsten). Sie können Ihre eigene `toString()` definieren:

```
// Constructors/Scientist.kt

class Scientist(val name: String) {
  override fun toString() =
    "Scientist('$name')"
}

fun main() {
  val zeep = Scientist("Zeep Xanflorp")
  println(zeep)
}
/* Output:
Scientist('Zeep Xanflorp')
*/
```

`override` ist ein neues Schlüsselwort für uns. Es ist hier erforderlich, weil `toString()` bereits eine Definition hat, die eine primitive Ausgabe erzeugt. `override` teilt Kotlin mit, dass wir tatsächlich die Standarddefinition von `toString()` durch unsere eigene Definition ersetzen möchten. Die Deutlichkeit von `override` macht den Code klarer und verhindert Fehler.

Ein `toString()`, das den Inhalt eines Objekts in einer praktischen Form anzeigt, ist nützlich, um Programmierfehler zu finden und zu beheben. Um den Prozess des *Debuggens* zu vereinfachen, bieten IDEs *Debugger*[21] an, die es Ihnen ermöglichen,

[21] https://www.jetbrains.com/help/idea/debugging-code.html

jeden Schritt der Programmausführung zu beobachten und in Ihre Objekte hinein zu sehen.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Einschränkung der Sichtbarkeit

> Wenn Sie ein Stück Code für ein paar Tage oder Wochen liegen lassen und dann zurückkehren, sehen Sie möglicherweise eine viel bessere Möglichkeit, es zu schreiben.

Dies ist eine der Hauptmotivationen für das *Refactoring*, das funktionierenden Code umschreibt, um ihn lesbarer, verständlicher und damit wartbarer zu machen.

Es gibt eine Spannung in diesem Wunsch, Ihren Code zu ändern und zu verbessern. Verbraucher (*Anwenderprogrammierer*) erfordern, dass Aspekte Ihres Codes stabil bleiben. Sie möchten es ändern, und sie wollen, dass es gleich bleibt.

Dies ist besonders wichtig für Bibliotheken. Verbraucher einer Bibliothek wollen nicht den Code für eine neue Version dieser Bibliothek umschreiben. Der Bibliotheksentwickler muss jedoch frei sein, Änderungen und Verbesserungen vorzunehmen, mit der Gewissheit, dass der Client-Code von diesen Änderungen nicht betroffen sein wird.

Daher ist eine primäre Überlegung im Softwaredesign:

> *Trenne Dinge, die sich ändern, von Dingen, die gleich bleiben.*

Um die Sichtbarkeit zu steuern, bieten Kotlin und einige andere Sprachen *Zugriffsmodifikatoren*. Bibliotheksentwickler entscheiden mit den Modifikatoren `public`, `private`, `protected` und `internal`, was für den Anwenderprogrammierer zugänglich ist und was nicht. Dieses Kapitel behandelt `public` und `private`, mit einer kurzen Einführung in `internal`. Wir erklären `protected` später im Buch.

Ein Zugriffsmodifikator wie `private` erscheint vor der Definition einer Klasse, Funktion oder Eigenschaft. Ein Zugriffsmodifikator steuert nur den Zugriff für diese spezielle Definition.

Eine `public` Definition ist für Anwenderprogrammierer zugänglich, sodass Änderungen an dieser Definition den Client-Code direkt beeinflussen. Wenn Sie keinen

Modifikator angeben, ist Ihre Definition automatisch `public`, daher ist `public` technisch gesehen redundant. Manchmal geben Sie dennoch `public` zur Klarstellung an.

Eine `private` Definition ist verborgen und nur von anderen Mitgliedern derselben Klasse zugänglich. Änderungen oder sogar das Entfernen einer `private` Definition beeinflussen die Anwenderprogrammierer nicht direkt.

`private` Klassen, Top-Level-Funktionen und Top-Level-Eigenschaften sind nur innerhalb dieser Datei zugänglich:

```
// Visibility/RecordAnimals.kt

private var index = 0                   // [1]

private class Animal(val name: String) // [2]

private fun recordAnimal(               // [3]
  animal: Animal
) {
  println("Animal #$index: ${animal.name}")
  index++
}

fun recordAnimals() {
  recordAnimal(Animal("Tiger"))
  recordAnimal(Animal("Antelope"))
}

fun recordAnimalsCount() {
  println("$index animals are here!")
}
```

Sie können auf `private` Top-Level-Eigenschaften (**[1]**), Klassen (**[2]**) und Funktionen (**[3]**) von anderen Funktionen und Klassen innerhalb von `RecordAnimals.kt` zugreifen. Kotlin verhindert, dass Sie auf ein `private` Top-Level-Element aus einer anderen Datei zugreifen, indem es Ihnen mitteilt, dass es in der Datei `private` ist:

```
// Visibility/ObserveAnimals.kt

fun main() {
  // Can't access private members
  // declared in another file.
  // Class is private:
  // val rabbit = Animal("Rabbit")
  // Function is private:
  // recordAnimal(rabbit)
  // Property is private:
  // index++

  recordAnimals()
  recordAnimalsCount()
}
/* Output:
Animal #0: Tiger
Animal #1: Antelope
2 animals are here!
*/
```

Sichtbarkeit wird am häufigsten für Mitglieder einer Klasse verwendet:

```
// Visibility/Cookie.kt

class Cookie(
  private var isReady: Boolean  // [1]
) {
  private fun crumble() =       // [2]
    println("crumble")

  public fun bite() =           // [3]
    println("bite")

  fun eat() {                   // [4]
    isReady = true              // [5]
    crumble()
    bite()
  }
}

fun main() {
```

```
  val x = Cookie(false)
  x.bite()
  // Can't access private members:
  // x.isReady
  // x.crumble()
  x.eat()
}
/* Output:
bite
crumble
bite
*/
```

- **[1]** Eine `private` Eigenschaft, die außerhalb der umgebenden Klasse nicht zugänglich ist.
- **[2]** Eine `private` Mitgliedsfunktion.
- **[3]** Eine `public` Mitgliedsfunktion, die für jeden zugänglich ist.
- **[4]** Kein Zugriffsmodifikator bedeutet `public`.
- **[5]** Nur Mitglieder derselben Klasse können auf `private` Mitglieder zugreifen.

Das Schlüsselwort `private` bedeutet, dass niemand auf dieses Mitglied zugreifen kann, außer anderen Mitgliedern dieser Klasse. Andere Klassen können nicht auf `private` Mitglieder zugreifen, sodass es so ist, als würden Sie die Klasse auch gegen sich selbst und Ihre Mitarbeiter abschirmen. Mit `private` können Sie dieses Mitglied nach Belieben ändern, ohne sich Sorgen machen zu müssen, ob es eine andere Klasse im selben Paket betrifft. Als Bibliotheksentwickler werden Sie typischerweise so viel wie möglich als `private` halten und nur Funktionen und Klassen für die Benutzerprogrammierer freigeben.

Jede Mitgliedsfunktion, die eine *Hilfsfunktion* für eine Klasse ist, kann `private` gemacht werden, um sicherzustellen, dass Sie sie nicht versehentlich anderswo im Paket verwenden und sich dadurch daran hindern, diese Funktion zu ändern oder zu entfernen.

Dasselbe gilt für eine `private` Eigenschaft innerhalb einer Klasse. Es sei denn, Sie müssen die zugrunde liegende Implementierung offenlegen (was weniger wahrscheinlich ist, als Sie vielleicht denken), machen Sie Eigenschaften `private`. Allerdings bedeutet eine `private` Referenz auf ein Objekt innerhalb einer Klasse nicht, dass ein anderes Objekt nicht eine `public` Referenz auf dasselbe Objekt haben kann:

```
// Visibility/MultipleRef.kt

class Counter(var start: Int) {
  fun increment() {
    start += 1
  }
  override fun toString() = start.toString()
}

class CounterHolder(counter: Counter) {
  private val ctr = counter
  override fun toString() =
    "CounterHolder: " + ctr
}

fun main() {
  val c = Counter(11)                  // [1]
  val ch = CounterHolder(c)            // [2]
  println(ch)
  c.increment()                        // [3]
  println(ch)
  val ch2 = CounterHolder(Counter(9)) // [4]
  println(ch2)
}
/* Output:
CounterHolder: 11
CounterHolder: 12
CounterHolder: 9
*/
```

- **[1]** `c` ist jetzt im Geltungsbereich definiert, der die Erstellung des `CounterHolder`-Objekts in der folgenden Zeile *umgibt*.
- **[2]** `c` als Argument an den `CounterHolder`-Konstruktor zu übergeben, bedeutet, dass der neue `CounterHolder` nun auf dasselbe `Counter`-Objekt verweist, auf das auch `c` verweist.
- **[3]** Der `Counter`, der angeblich `privat` innerhalb von `ch` ist, kann dennoch über `c` manipuliert werden.
- **[4]** `Counter(9)` hat keine anderen Referenzen außer innerhalb von `CounterHolder`, daher kann es nicht von etwas anderem als `ch2` zugegriffen oder modifiziert werden.

Mehrere Referenzen auf ein einzelnes Objekt zu haben, wird als *Aliasing* bezeichnet und kann überraschendes Verhalten hervorrufen.

Module

Im Gegensatz zu den kleinen Beispielen in diesem Buch sind reale Programme oft groß. Es kann hilfreich sein, solche Programme in ein oder mehrere *Module* zu unterteilen. Ein Modul ist ein logisch unabhängiger Teil eines Codebestands. Die Art und Weise, wie Sie ein Projekt in Module unterteilen, hängt vom Build-System ab (wie Gradle[22] oder Maven[23]) und liegt außerhalb des Rahmens dieses Buches.

Eine `interne` Definition ist nur innerhalb des Moduls zugänglich, in dem sie definiert ist. `Intern` liegt irgendwo zwischen `privat` und `öffentlich`—verwenden Sie es, wenn `privat` zu restriktiv ist, Sie aber nicht möchten, dass ein Element Teil der `öffentlichen` API ist. Wir verwenden `intern` nicht in den Beispielen oder Übungen des Buches.

Module sind ein höheres Konzept. Der folgende Abschnitt führt *Pakete* ein, die eine feinere Strukturierung ermöglichen. Eine Bibliothek ist oft ein einziges Modul, das aus mehreren Paketen besteht, sodass `interne` Elemente innerhalb der Bibliothek verfügbar sind, jedoch nicht von den Verbrauchern dieser Bibliothek zugänglich sind.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

[22] https://gradle.org/
[23] https://maven.apache.org/

Pakete

> Ein grundlegendes Prinzip in der Programmierung ist das Akronym DRY: *Wiederhole dich nicht.*

Mehrere identische Codefragmente erfordern Wartung, wann immer Sie Korrekturen oder Verbesserungen vornehmen. Das Duplizieren von Code ist also nicht nur zusätzliche Arbeit—jede Duplikation bietet Chancen für Fehler.

Das Schlüsselwort `import` wiederverwendet Code aus anderen Dateien. Eine Möglichkeit, `import` zu verwenden, besteht darin, einen Klassen-, Funktions- oder Eigenschaftsnamen anzugeben:

```
import packagename.ClassName
import packagename.functionName
import packagename.propertyName
```

Ein *Paket* ist eine zugehörige Sammlung von Code. Jedes Paket ist normalerweise dafür ausgelegt, ein bestimmtes Problem zu lösen, und enthält oft mehrere Funktionen und Klassen. Zum Beispiel können wir mathematische Konstanten und Funktionen aus der `kotlin.math` Bibliothek importieren:

```
// Packages/ImportClass.kt
import kotlin.math.PI
import kotlin.math.cos  // Cosine

fun main() {
  println(PI)
  println(cos(PI))
  println(cos(2 * PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

Manchmal möchte man mehrere Drittanbieter-Bibliotheken verwenden, die Klassen oder Funktionen mit demselben Namen enthalten. Das Schlüsselwort `as` ermöglicht es Ihnen, beim Importieren Namen zu ändern:

```
// Packages/ImportNameChange.kt
import kotlin.math.PI as circleRatio
import kotlin.math.cos as cosine

fun main() {
  println(circleRatio)
  println(cosine(circleRatio))
  println(cosine(2 * circleRatio))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

`as` ist nützlich, wenn der Bibliotheksname schlecht gewählt oder übermäßig lang ist.

Sie können einen Import im Hauptteil Ihres Codes vollständig qualifizieren. Im folgenden Beispiel könnte der Code aufgrund der expliziten Paketnamen weniger lesbar sein, aber die Herkunft jedes Elements ist absolut klar:

```
// Packages/FullyQualify.kt

fun main() {
  println(kotlin.math.PI)
  println(kotlin.math.cos(kotlin.math.PI))
  println(kotlin.math.cos(2 * kotlin.math.PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

Um alles aus einem Paket zu importieren, verwenden Sie einen Stern:

```
// Packages/ImportEverything.kt
import kotlin.math.*

fun main() {
  println(E)
  println(E.roundToInt())
  println(E.toInt())
}
/* Output:
2.718281828459045
3
2
*/
```

Das `kotlin.math`-Paket enthält eine praktische Funktion `roundToInt()`, die den `Double`-Wert auf die nächste ganze Zahl aufrundet, im Gegensatz zu `toInt()`, das einfach alles nach einem Dezimalpunkt abschneidet.

Um Ihren Code wiederzuverwenden, erstellen Sie ein Paket mit dem Schlüsselwort `package`. Die `package`-Anweisung muss die erste nicht-kommentare Anweisung in der Datei sein. `package` wird gefolgt vom Namen Ihres Pakets, der konventionell komplett in Kleinbuchstaben geschrieben wird:

```
// Packages/PythagoreanTheorem.kt
package pythagorean
import kotlin.math.sqrt

class RightTriangle(
  val a: Double,
  val b: Double
) {
  fun hypotenuse() = sqrt(a * a + b * b)
  fun area() = a * b / 2
}
```

Sie können die Quellcodedatei beliebig benennen, im Gegensatz zu Java, das erfordert, dass der Dateiname mit dem Klassennamen identisch ist.

Kotlin erlaubt Ihnen, einen beliebigen Namen für Ihr Paket zu wählen, aber es wird als guter Stil betrachtet, wenn der Paketname mit dem Verzeichnisnamen identisch ist, in dem sich die Paketdateien befinden (dies wird nicht immer der Fall für die Beispiele in diesem Buch sein).

Die Elemente im pythagorean Paket sind jetzt mit import verfügbar:

```
// Packages/ImportPythagorean.kt
import pythagorean.RightTriangle

fun main() {
  val rt = RightTriangle(3.0, 4.0)
  println(rt.hypotenuse())
  println(rt.area())
}
/* Output:
5.0
6.0
*/
```

Im Rest dieses Buches verwenden wir package-Anweisungen für jede Datei, die Funktionen, Klassen usw. außerhalb von main() definiert, um Namenskonflikte mit anderen Dateien im Buch zu vermeiden. In der Regel werden wir jedoch keine package-Anweisung in einer Datei platzieren, die *nur* ein main() enthält.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Testen

> Konstantes Testen ist unerlässlich für eine schnelle Programmentwicklung.

Wenn das Ändern eines Teils Ihres Codes anderen Code beschädigt, zeigen Ihre Tests das Problem sofort auf. Wenn Sie es nicht sofort herausfinden, häufen sich die Änderungen an und Sie können nicht mehr feststellen, welche Änderung das Problem verursacht hat. Sie werden *viel* länger brauchen, um es zu verfolgen.

Testen ist eine entscheidende Praxis, daher führen wir es früh ein und verwenden es im gesamten Rest des Buches. Auf diese Weise gewöhnen Sie sich daran, Tests als Standardteil des Programmierprozesses zu betrachten.

`println()` zu verwenden, um die Korrektheit des Codes zu überprüfen, ist ein schwacher Ansatz—Sie müssen jedes Mal die Ausgabe genau prüfen und bewusst sicherstellen, dass sie korrekt ist.

Um Ihre Erfahrung beim Verwenden dieses Buches zu vereinfachen, haben wir unser eigenes kleines Testsystem erstellt. Das Ziel ist ein minimaler Ansatz, der:

1. Das erwartete Ergebnis von Ausdrücken zeigt.
2. Eine Ausgabe liefert, damit Sie wissen, dass das Programm läuft, selbst wenn alle Tests erfolgreich sind.
3. Das Konzept des Testens frühzeitig in Ihrer Praxis verankert.

Obwohl es für dieses Buch nützlich ist, ist unseres *kein* Testsystem für den Arbeitsplatz. Andere haben lange und hart gearbeitet, um solche Testsysteme zu erstellen. Zum Beispiel:

- JUnit[24] ist eines der beliebtesten Java-Testframeworks und kann leicht in Kotlin verwendet werden.
- Kotest[25] ist speziell für Kotlin entwickelt und nutzt die Funktionen der Kotlin-Sprache.

[24] https://junit.org
[25] https://github.com/kotest/kotest

- Das Spek Framework[26] produziert eine andere Form des Testens, genannt *Spezifikationstests*.

Um unser Testframework zu verwenden, müssen wir es zuerst `importieren`. Die grundlegenden Elemente des Frameworks sind `eq` (*gleich*) und `neq` (*nicht gleich*):

```
// Testing/TestingExample.kt
import atomictest.*

fun main() {
  val v1 = 11
  val v2 = "Ontology"

  // 'eq' means "equals":
  v1 eq 11
  v2 eq "Ontology"

  // 'neq' means "not equal"
  v2 neq "Epistemology"

  // [Error] Epistemology != Ontology
  // v2 eq "Epistemology"
}
/* Output:
11
Ontology
Ontology
*/
```

Der Code für das Paket `atomictest` befindet sich in Appendix A: AtomicTest. Es ist nicht beabsichtigt, dass Sie alles in `AtomicTest.kt` sofort verstehen, da es einige Funktionen verwendet, die erst später im Buch erscheinen werden.

Um ein klares, angenehmes Erscheinungsbild zu erzeugen, verwendet `AtomicTest` eine Kotlin-Funktion, die Sie noch nicht gesehen haben: die Fähigkeit, einen Funktionsaufruf `a.function(b)` in der textähnlichen Form `a function b` zu schreiben. Dies wird als *Infix-Notation* bezeichnet. Nur Funktionen, die mit dem Schlüsselwort `infix` definiert sind, können auf diese Weise aufgerufen werden. `AtomicTest.kt` definiert die `infix`-Funktionen `eq` und `neq`, die in `TestingExample.kt` verwendet werden:

[26] https://spekframework.org/

```
expression eq expected
expression neq expected
```

eq und neq sind flexibel – fast alles funktioniert als Testausdruck. Wenn *erwartet* ein String ist, wird *Ausdruck* in einen String umgewandelt und die beiden Strings werden verglichen. Andernfalls werden *Ausdruck* und *erwartet* direkt verglichen (ohne sie vorher umzuwandeln). In jedem Fall erscheint das Ergebnis von *Ausdruck* auf der Konsole, sodass Sie etwas sehen, wenn das Programm läuft. Selbst wenn die Tests erfolgreich sind, sehen Sie das Ergebnis links von eq oder neq. Wenn *Ausdruck* und *erwartet* nicht gleichwertig sind, zeigt AtomicTest einen Fehler an, wenn das Programm läuft.

Der letzte Test in TestingExample.kt schlägt absichtlich fehl, damit Sie ein Beispiel für eine Fehlerausgabe sehen. Wenn die beiden Werte nicht gleich sind, zeigt Kotlin die entsprechende Nachricht an, die mit [Error] beginnt. Wenn Sie die letzte Zeile auskommentieren und das obige Beispiel ausführen, sehen Sie nach allen erfolgreichen Tests:

```
[Error] Epistemology != Ontology
```

Der tatsächliche Wert, der in v2 gespeichert ist, entspricht nicht dem, was im Ausdruck "erwartet" behauptet wird. AtomicTest zeigt die String-Darstellungen sowohl für erwartete als auch für tatsächliche Werte an.

eq und neq sind die grundlegenden (infix) Funktionen, die für AtomicTest definiert sind – es ist wirklich ein minimalistisches Testsystem. Wenn Sie eq- und neq-Ausdrücke in Ihren Beispielen verwenden, erstellen Sie sowohl einen Test als auch eine Konsolenausgabe. Sie überprüfen die Korrektheit des Programms, indem Sie es ausführen.

Es gibt ein zweites Werkzeug in AtomicTest. Das trace-Objekt erfasst die Ausgabe für einen späteren Vergleich:

```
// Testing/Trace1.kt
import atomictest.*

fun main() {
  trace("line 1")
  trace(47)
  trace("line 2")
  trace eq """
    line 1
    47
    line 2
  """
}
```

Das Hinzufügen von Ergebnissen zu `trace` sieht aus wie ein Funktionsaufruf, daher können Sie `println()` effektiv durch `trace()` ersetzen.

In früheren Atomen haben wir die Ausgabe angezeigt und uns auf die menschliche visuelle Inspektion verlassen, um Unstimmigkeiten zu erkennen. Das ist unzuverlässig; selbst in einem Buch, in dem wir den Code immer wieder genau prüfen, haben wir gelernt, dass man der visuellen Inspektion nicht trauen kann, um Fehler zu finden. Von nun an verwenden wir selten kommentierte Ausgabeblöcke, da `AtomicTest` alles für uns erledigen wird. Manchmal fügen wir jedoch immer noch kommentierte Ausgabeblöcke ein, wenn dies einen nützlicheren Effekt hat.

Die Vorteile des Testens im gesamten restlichen Buch sollten Ihnen helfen, das Testen in Ihren Programmierprozess zu integrieren. Sie werden sich wahrscheinlich unwohl fühlen, wenn Sie Code sehen, der keine Tests hat. Sie könnten sogar entscheiden, dass Code ohne Tests per Definition fehlerhaft ist.

Testen als Teil der Programmierung

Testen ist am effektivsten, wenn es in Ihren Softwareentwicklungsprozess integriert ist. Das Schreiben von Tests stellt sicher, dass Sie die erwarteten Ergebnisse erhalten. Viele Leute befürworten das Schreiben von Tests *vor* dem Schreiben des Implementierungscodes - Sie lassen zuerst den Test fehlschlagen, bevor Sie den Code schreiben, um ihn erfolgreich zu machen. Diese Technik, genannt *Testgetriebene Entwicklung* (TDD), ist eine Möglichkeit sicherzustellen, dass Sie wirklich das testen, was Sie

denken. Eine vollständigere Beschreibung von TDD finden Sie auf Wikipedia (suchen Sie nach “Testgetriebene Entwicklung”).

Es gibt einen weiteren Vorteil beim testbaren Schreiben - es verändert die Art und Weise, wie Sie Ihren Code gestalten. Sie könnten die Ergebnisse einfach auf der Konsole anzeigen. Aber im Testdenken fragen Sie sich: “Wie werde ich das testen?” Wenn Sie eine Funktion erstellen, entscheiden Sie, dass Sie etwas aus der Funktion zurückgeben sollten, wenn auch nur, um dieses Ergebnis zu testen. Funktionen, die nichts anderes tun, als Eingaben zu nehmen und Ausgaben zu erzeugen, neigen dazu, auch bessere Designs zu erzeugen.

Hier ist ein vereinfachtes Beispiel, das TDD verwendet, um die BMI-Berechnung aus Zahlentypen zu implementieren. Zuerst schreiben wir die Tests sowie eine anfängliche Implementierung, die fehlschlägt (weil wir die Funktionalität noch nicht implementiert haben):

```
// Testing/TDDFail.kt
package testing1
import atomictest.eq

fun main() {
  calculateBMI(160, 68) eq "Normal weight"
//  calculateBMI(100, 68) eq "Underweight"
//  calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(lbs: Int, height: Int) =
  "Normal weight"
```

Nur der erste Test besteht. Die anderen Tests schlagen fehl und sind kommentiert. Als nächstes fügen wir Code hinzu, um zu bestimmen, welche Gewichte in welchen Kategorien sind. Jetzt schlagen *alle* Tests fehl:

```
// Testing/TDDStillFails.kt
package testing2
import atomictest.eq

fun main() {
  // Everything fails:
  // calculateBMI(160, 68) eq "Normal weight"
  // calculateBMI(100, 68) eq "Underweight"
  // calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(
  lbs: Int,
  height: Int
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

Wir verwenden `Int`s anstelle von `Double`s, was zu einem Nullergebnis führt. Die Tests führen uns zur Lösung:

```
// Testing/TDDWorks.kt
package testing3
import atomictest.eq

fun main() {
  calculateBMI(160.0, 68.0) eq "Normal weight"
  calculateBMI(100.0, 68.0) eq "Underweight"
  calculateBMI(200.0, 68.0) eq "Overweight"
}

fun calculateBMI(
  lbs: Double,
  height: Double
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

Sie können zusätzliche Tests für die Randbedingungen hinzufügen.

In den Übungen für dieses Buch haben wir Tests enthalten, die Ihr Code bestehen muss.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Ausnahmen

> Das Wort “Ausnahme” wird im gleichen Sinne verwendet wie der Ausdruck “Ich nehme Anstoß daran.”

Eine außergewöhnliche Bedingung verhindert die Fortsetzung der aktuellen Funktion oder des aktuellen Bereichs. An dem Punkt, an dem das Problem auftritt, wissen Sie möglicherweise nicht, was Sie damit tun sollen, aber Sie können im aktuellen Kontext nicht fortfahren. Sie haben nicht genügend Informationen, um das Problem zu beheben. Daher müssen Sie stoppen und das Problem an einen anderen Kontext übergeben, der geeignete Maßnahmen ergreifen kann.

Dieses Atom behandelt die Grundlagen von *Ausnahmen* als ein Mechanismus zur Fehlerberichterstattung. In Abschnitt VI: Fehlervermeidung betrachten wir andere Möglichkeiten, mit Problemen umzugehen.

Es ist wichtig, eine außergewöhnliche Bedingung von einem normalen Problem zu unterscheiden. Ein normales Problem verfügt über genügend Informationen im aktuellen Kontext, um das Problem zu bewältigen. Bei einer außergewöhnlichen Bedingung können Sie die Verarbeitung nicht fortsetzen. Alles, was Sie tun können, ist zu gehen und das Problem einem externen Kontext zu überlassen. Dies ist der Fall, wenn Sie *eine Ausnahme werfen*. Die Ausnahme ist das Objekt, das vom Ort des Fehlers “geworfen” wird.

Betrachten Sie `toInt()`, das einen `String` in einen `Int` umwandelt. Was passiert, wenn Sie diese Funktion für einen `String` aufrufen, der keinen ganzzahligen Wert enthält?

```
// Exceptions/ToIntException.kt
package exceptions

fun erroneousCode() {
  // Uncomment this line to get an exception:
  // val i = "1$".toInt()        // [1]
}

fun main() {
  erroneousCode()
}
```

Das Auskommentieren der Zeile **[1]** führt zu einer Ausnahme. Hier ist die fehlerhafte Zeile kommentiert, damit der Aufbau des Buches nicht gestoppt wird, das überprüft, ob jedes Beispiel wie erwartet kompiliert und ausgeführt wird.

Wenn eine Ausnahme ausgelöst wird, stoppt der Ausführungspfad—derjenige, der nicht fortgesetzt werden kann—und das Ausnahmeobjekt wird aus dem aktuellen Kontext herausgeworfen. Hier verlässt es den Kontext von `erroneousCode()` und geht in den Kontext von `main()`. In diesem Fall meldet Kotlin nur den Fehler; der Programmierer hat vermutlich einen Fehler gemacht und muss den Code korrigieren.

Wenn eine Ausnahme nicht abgefangen wird, bricht das Programm ab und zeigt einen *Stack-Trace* mit detaillierten Informationen an. Das Auskommentieren der Zeile **[1]** in `ToIntException.kt` führt zu folgendem Output:

```
Exception in thread "main" java.lang.NumberFormatException: For input s\
tring: "1$"
  at java.lang.NumberFormatException.forInputString(NumberFormatExcepti\
on.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at ToIntExceptionKt.erroneousCode(at ToIntException.kt:6)
  at ToIntExceptionKt.main(at ToIntException.kt:10)
```

Der Stack-Trace liefert Details wie die Datei und die Zeile, in der die Ausnahme aufgetreten ist, sodass Sie das Problem schnell entdecken können. Die letzten beiden Zeilen zeigen das Problem: In Zeile 10 von `main()` rufen wir `erroneousCode()` auf. Dann, genauer gesagt, in Zeile 6 von `erroneousCode()` rufen wir `toInt()` auf.

Um das Kommentieren und Auskommentieren von Code zur Anzeige von Ausnahmen zu vermeiden, verwenden wir die Funktion `capture()` aus dem Paket `AtomicTest`:

```
// Exceptions/IntroducingCapture.kt
import atomictest.*

fun main() {
  capture {
    "1$".toInt()
  } eq "NumberFormatException: " +
    """For input string: "1$""""
}
```

Mit `capture()` vergleichen wir die generierte Ausnahme mit der erwarteten Fehlermeldung. `capture()` ist nicht sehr hilfreich für normale Programmierung - es ist speziell für dieses Buch entworfen, damit Sie die Ausnahme sehen und wissen können, dass die Ausgabe vom Build-System des Buches überprüft wurde.

Eine weitere Strategie, wenn Sie das erwartete Ergebnis nicht erfolgreich erzielen können, besteht darin, `null` zurückzugeben, eine spezielle Konstante, die "kein Wert" bedeutet. Sie können `null` anstelle eines Wertes jeden Typs zurückgeben. Später in Nullable Typen besprechen wir, wie `null` den Typ des resultierenden Ausdrucks beeinflusst.

Die Kotlin-Standardbibliothek enthält `String.toIntOrNull()`, das die Umwandlung durchführt, wenn der `String` eine ganze Zahl enthält, oder `null` produziert, wenn die Umwandlung unmöglich ist - `null` ist eine einfache Möglichkeit, einen Fehler anzuzeigen:

```
// Exceptions/IntroducingNull.kt
import atomictest.eq

fun main() {
  "1$".toIntOrNull() eq null
}
```

Angenommen, wir berechnen das durchschnittliche Einkommen über einen Zeitraum von Monaten:

```
// Exceptions/AverageIncome.kt
package firstversion
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "ArithmeticException: / by zero"
}
```

Wenn `months` null ist, wirft die Division in `averageIncome()` eine `ArithmeticException`. Leider sagt uns dies nichts darüber, warum der Fehler aufgetreten ist, was der Nenner bedeutet und ob er überhaupt null sein darf. Dies ist eindeutig ein Fehler im Code—`averageIncome()` sollte mit einem `months` von `0` so umgehen, dass ein Division durch Null Fehler vermieden wird.

Lassen Sie uns `averageIncome()` modifizieren, um mehr Informationen über die Quelle des Problems zu liefern. Wenn `months` null ist, können wir keinen normalen Ganzzahlwert als Ergebnis zurückgeben. Eine Strategie ist es, `null` zurückzugeben:

```
// Exceptions/AverageIncomeWithNull.kt
package withnull
import atomictest.eq

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    null
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  averageIncome(5000, 0) eq null
}
```

Wenn eine Funktion `null` zurückgeben kann, verlangt Kotlin, dass Sie das Ergebnis überprüfen, bevor Sie es verwenden (dies wird in Nullable Typen behandelt). Selbst

wenn Sie nur dem Benutzer eine Ausgabe anzeigen möchten, ist es besser zu sagen: "Es sind keine vollen Monatszeiträume vergangen," anstatt "Ihr durchschnittliches Einkommen für den Zeitraum ist: null."

Anstatt `averageIncome()` mit den falschen Argumenten auszuführen, können Sie eine Ausnahme auslösen – entkommen und einen anderen Teil des Programms zwingen, das Problem zu verwalten. Sie *könnten* die Standard-`ArithmeticException` zulassen, aber es ist oft nützlicher, eine spezifische Ausnahme mit einer detaillierten Fehlermeldung zu werfen. Wenn Ihre Anwendung nach ein paar Jahren im Einsatz plötzlich eine Ausnahme auslöst, weil eine neue Funktion `averageIncome()` aufruft, ohne die Argumente richtig zu überprüfen, werden Sie für diese Nachricht dankbar sein:

```
// Exceptions/AverageIncomeWithException.kt
package properexception
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    throw IllegalArgumentException(    // [1]
      "Months can't be zero")
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "IllegalArgumentException: " +
    "Months can't be zero"
}
```

- **[1]** Beim Auslösen einer Ausnahme wird das Schlüsselwort `throw` gefolgt von der Ausnahme, die ausgelöst werden soll, zusammen mit allen Argumenten, die sie möglicherweise benötigt. Hier verwenden wir die Standard-Ausnahmeklasse `IllegalArgumentException`.

Ihr Ziel ist es, die nützlichsten Nachrichten zu generieren, um die Unterstützung Ihrer Anwendung in Zukunft zu vereinfachen. Später lernen Sie, Ihre eigenen Ausnahme-Typen zu definieren und sie spezifisch auf Ihre Umstände abzustimmen.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Listen

> Eine `List` ist ein *Behälter*, also ein Objekt, das andere Objekte enthält.

Behälter werden auch als *Sammlungen* bezeichnet. Wenn wir einen grundlegenden Behälter für die Beispiele in diesem Buch benötigen, verwenden wir normalerweise eine `List`.

`Lists` sind Teil des Standard-Kotlin-Pakets, daher benötigen sie keinen `import`.

Das folgende Beispiel erstellt eine `List`, die mit `Int`s gefüllt ist, indem die Standardbibliotheksfunktion `listOf()` mit Initialisierungswerten aufgerufen wird:

```
// Lists/Lists.kt
import atomictest.eq

fun main() {
  val ints = listOf(99, 3, 5, 7, 11, 13)
  ints eq "[99, 3, 5, 7, 11, 13]"   // [1]

  // Select each element in the List:
  var result = ""
  for (i in ints) {                 // [2]
    result += "$i "
  }
  result eq "99 3 5 7 11 13"

  // "Indexing" into the List:
  ints[4] eq 11                     // [3]
}
```

- **[1]** Eine `List` verwendet eckige Klammern, um sich selbst darzustellen.
- **[2]** `for`-Schleifen funktionieren gut mit `Lists`: `for(i in ints)` bedeutet, dass `i` jeden Wert in `ints` erhält. Sie deklarieren `val i` nicht und geben auch nicht seinen Typ an; Kotlin erkennt aus dem Kontext, dass `i` ein `for`-Schleifen-Identifikator ist.

- **[3]** Eckige Klammern *indexieren* in eine `List`. Eine `List` behält ihre Elemente in der Initialisierungsreihenfolge bei, und Sie wählen sie einzeln nach Nummer aus. Wie in den meisten Programmiersprachen beginnt Kotlin das Indexieren beim Element Null, was in diesem Fall den Wert `99` ergibt. Somit ergibt ein Index von `4` den Wert `11`.

Das Vergessen, dass das Indexieren bei Null beginnt, führt zum sogenannten *Eins-zu-viel-Fehler*. In einer Sprache wie Kotlin wählen wir oft nicht Elemente einzeln aus, sondern *iterieren* stattdessen durch einen gesamten Container mit `in`. Dies eliminiert Eins-zu-viel-Fehler.

Wenn Sie einen Index über das letzte Element in einer `List` hinaus verwenden, wirft Kotlin eine `ArrayIndexOutOfBoundsException`:

```
// Lists/OutOfBounds.kt
import atomictest.*

fun main() {
  val ints = listOf(1, 2, 3)
  capture {
    ints[3]
  } contains
    listOf("ArrayIndexOutOfBoundsException")
}
```

Eine `List` kann alle verschiedenen Typen halten. Hier ist eine `List` von `Doubles` und eine `List` von `Strings`:

```
// Lists/ListUsefulFunction.kt
import atomictest.eq

fun main() {
  val doubles =
    listOf(1.1, 2.2, 3.3, 4.4)
  doubles.sum() eq 11.0

  val strings = listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings eq listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings.sorted() eq listOf("And",
```

```
    "Brillig", "Slithy", "Toves", "Twas")
  strings.reversed() eq listOf("Toves",
    "Slithy", "And", "Brillig", "Twas")
  strings.first() eq "Twas"
  strings.takeLast(2) eq
    listOf("Slithy", "Toves")
}
```

Dies zeigt einige der Operationen von `List`. Beachten Sie den Namen "sorted" anstelle von "sort". Wenn Sie `sorted()` aufrufen, *erzeugt* es eine neue `List`, die die gleichen Elemente wie die alte in sortierter Reihenfolge enthält—aber es lässt die ursprüngliche `List` unverändert. Es "sort" zu nennen, impliziert, dass die ursprüngliche `List` direkt verändert wird (auch bekannt als *sortiert an Ort und Stelle*). In Kotlin sieht man häufig diese Tendenz, "das ursprüngliche Objekt unverändert zu lassen und ein neues Objekt zu erzeugen." `reversed()` erzeugt ebenfalls eine neue `List`.

Parametrisierte Typen

Wir betrachten es als gute Praxis, Typinferenz zu verwenden—es neigt dazu, den Code sauberer und leichter lesbar zu machen. Manchmal jedoch beschwert sich Kotlin, dass es nicht herausfinden kann, welchen Typ es verwenden soll, und in anderen Fällen macht Explizitheit den Code verständlicher. So teilen wir Kotlin mit, welchen Typ eine `List` enthält:

```
// Lists/ParameterizedTypes.kt
import atomictest.eq

fun main() {
  // Type is inferred:
  val numbers = listOf(1, 2, 3)
  val strings =
    listOf("one", "two", "three")
  // Exactly the same, but explicitly typed:
  val numbers2: List<Int> = listOf(1, 2, 3)
  val strings2: List<String> =
    listOf("one", "two", "three")
  numbers eq numbers2
```

```
  strings eq strings2
}
```

Kotlin verwendet die Initialisierungswerte, um abzuleiten, dass `numbers` eine `List` von `Int`s enthält, während `strings` eine `List` von `String`s enthält.

`numbers2` und `strings2` sind explizit typisierte Versionen von `numbers` und `strings`, erstellt durch das Hinzufügen der Typdeklarationen `List<Int>` und `List<String>`. Sie haben Winkelklammern noch nicht gesehen - sie kennzeichnen einen *Typparameter*, der es Ihnen ermöglicht zu sagen: "Dieser Container enthält 'Parameter'-Objekte." Wir sprechen `List<Int>` als "`List` von `Int`" aus.

Typparameter sind nützlich für Komponenten, die keine Container sind, aber man sieht sie oft bei containerähnlichen Objekten.

Rückgabewerte können ebenfalls Typparameter haben:

```
// Lists/ParameterizedReturn.kt
package lists
import atomictest.eq

// Return type is inferred:
fun inferred(p: Char, q: Char) =
  listOf(p, q)

// Explicit return type:
fun explicit(p: Char, q: Char): List<Char> =
  listOf(p, q)

fun main() {
  inferred('a', 'b') eq "[a, b]"
  explicit('y', 'z') eq "[y, z]"
}
```

Kotlin leitet den Rückgabetyp für `inferred()` ab, während `explicit()` den Rückgabetyp der Funktion angibt. Man kann nicht einfach sagen, dass es eine `List` zurückgibt; Kotlin wird beanstanden, also muss man auch den Typ-Parameter angeben. Wenn Sie den Rückgabetyp einer Funktion angeben, setzt Kotlin Ihre Absicht durch.

Schreibgeschützte und veränderbare Listen

Wenn Sie nicht ausdrücklich sagen, dass Sie eine veränderbare `List` möchten, erhalten Sie keine. `listOf()` erzeugt eine schreibgeschützte `List`, die keine veränderbaren Funktionen hat.

Wenn Sie eine `List` schrittweise erstellen (das heißt, Sie haben nicht alle Elemente zum Zeitpunkt der Erstellung), verwenden Sie `mutableListOf()`. Dies erzeugt eine `MutableList`, die verändert werden kann:

```
// Lists/MutableList.kt
import atomictest.eq

fun main() {
  val list = mutableListOf<Int>()

  list.add(1)
  list.addAll(listOf(2, 3))

  list += 4
  list += listOf(5, 6)

  list eq listOf(1, 2, 3, 4, 5, 6)
}
```

Da `list` keine anfänglichen Elemente hat, müssen wir Kotlin mitteilen, welchen Typ es hat, indem wir die `<Int>`-Spezifikation im Aufruf von `mutableListOf()` angeben. Sie können Elemente zu einer `MutableList` mit `add()` und `addAll()` hinzufügen oder den Operator `+=` verwenden, der entweder ein einzelnes Element oder eine andere Sammlung hinzufügt.

Eine `MutableList` kann als `List` behandelt werden, in diesem Fall kann sie nicht geändert werden. Sie können jedoch eine schreibgeschützte `List` nicht als `MutableList` behandeln:

```
// Lists/MutListIsList.kt
package lists
import atomictest.eq

fun makeList(): List<Int> =
  mutableListOf(1, 2, 3)

fun main() {
  // makeList() produces a read-only List:
  val list = makeList()
  // list.add(3) // Unresolved reference: add
  list eq listOf(1, 2, 3)
}
```

`list` fehlt es an Mutationsfunktionen, obwohl es ursprünglich mit `mutableListOf()` innerhalb von `makeList()` erstellt wurde. Beachten Sie, dass der Ergebnistyp von `makeList()` `List<Int>` ist. Das ursprüngliche Objekt ist immer noch eine `MutableList`, wird aber durch die Linse einer `List` betrachtet.

Eine `List` ist *schreibgeschützt*—Sie können ihren Inhalt lesen, aber nicht schreiben. Wenn die zugrunde liegende Implementierung eine `MutableList` ist und Sie eine veränderbare Referenz auf diese Implementierung beibehalten, können Sie sie weiterhin über diese veränderbare Referenz modifizieren, und alle schreibgeschützten Referenzen werden diese Änderungen sehen. Dies ist ein weiteres Beispiel für *Aliasing*, eingeführt in Einschränken der Sichtbarkeit:

```
// Lists/MultipleListRefs.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first.add(2)
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first` ist eine unveränderliche Referenz (`val`) auf das veränderliche Objekt, das von `mutableListOf(1)` erzeugt wird. Wenn `second` auf `first` aliasiert wird, wird es zu einer Ansicht desselben Objekts. `second` ist schreibgeschützt, weil `List<Int>` keine

Änderungsfunktionen beinhaltet. Ohne die explizite `List<Int>` Typdeklaration würde Kotlin annehmen, dass `second` ebenfalls eine Referenz auf ein veränderliches Objekt ist.

Wir können dem Objekt ein Element (2) hinzufügen, weil `first` eine Referenz auf eine veränderliche `Liste` ist. Beachten Sie, dass `second` diese Änderungen beobachtet—es kann die `Liste` nicht ändern, obwohl die `Liste` über `first` geändert wird.

Das += Rätsel

Der += Operator kann den Anschein erwecken, dass eine unveränderliche `Liste` tatsächlich veränderlich ist:

```
// Lists/ApparentlyMutableList.kt
import atomictest.eq

fun main() {
  var list = listOf('X') // Immutable
  list += 'Y' // Appears to be mutable
  list eq "[X, Y]"
}
```

`listOf()` erzeugt eine unveränderliche `List`, aber `list += 'Y'` scheint diese `List` zu ändern. Verstößt += irgendwie gegen die Unveränderlichkeit?

Dies passiert nur, weil `list` ein `var` ist. Hier ist ein detaillierteres Beispiel, das die verschiedenen Kombinationen von veränderlichen/unveränderlichen `Lists` mit `val`/`var` zeigt:

```
// Lists/PlusAssignPuzzle.kt
import atomictest.eq

fun main() {
    // Mutable List assigned to a 'val'/'var':
    val list1 = mutableListOf('A') // or 'var'
    list1 += 'A' // Is the same as:
    list1.plusAssign('A')              // [1]

    // Immutable List assigned to a 'val':
    val list2 = listOf('B')
    // list2 += 'B' // Is the same as:
    // list2 = list2 + 'B'             // [2]

    // Immutable List assigned to a 'var':
    var list3 = listOf('C')
    list3 += 'C' // Is the same as:
    val newList = list3 + 'C'          // [3]
    list3 = newList                    // [4]

    list1 eq "[A, A, A]"
    list2 eq "[B]"
    list3 eq "[C, C, C]"
}
```

- **[1]** `list1` bezieht sich auf ein veränderbares Objekt, das daher vor Ort modifiziert werden kann. Der Compiler übersetzt `+=` zum Aufruf von `plusAssign()`. Es spielt keine Rolle, ob `list1` ein `val` oder ein `var` ist, da `list1` nach der Erstellung niemals *neu zugewiesen* wird—es verweist immer auf die gleiche veränderbare Liste. Wenn man es zu einem `var` macht, weist IntelliJ darauf hin, dass es sich nie ändert und schlägt vor, es zu einem `val` zu machen.
- **[2]** Dies versucht, eine neue `List` zu erstellen, indem `list2` und `'B'` kombiniert werden, aber es kann diese neue `List` nicht `list2` neu zuweisen, da `list2` ein `val` ist. Ohne die Möglichkeit, diese Neuzuweisung durchzuführen, kann `+=` nicht kompiliert werden.
- **[3]** Erstellt `newList` ohne die bestehende unveränderliche `List` zu modifizieren, auf die `list3` verweist.
- **[4]** Da `list3` ein `var` ist, weist der Compiler `newList` zurück in `list3` zu. Der vorherige Inhalt von `list3` wird dann vergessen, und es erscheint, als ob `list3`

verändert wurde. Tatsächlich wurde das alte `list3` verworfen und durch das neu erstellte `newList` ersetzt, was die Illusion erzeugt, dass `list3` veränderbar ist.

Dieses Verhalten von `+=` tritt auch bei anderen Sammlungen auf. Die daraus resultierende Verwirrung ist ein weiterer Grund, `val` gegenüber `var` für Ihre Bezeichner zu bevorzugen.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Variable Argumentlisten

> Das Schlüsselwort `vararg` erzeugt eine flexibel dimensionierte Argumentliste.

In Listen haben wir `listOf()` eingeführt, das eine beliebige Anzahl von Parametern akzeptiert und eine `Liste` erzeugt:

```
// Varargs/ListOf.kt
import atomictest.eq

fun main() {
  listOf(1) eq "[1]"
  listOf("a", "b") eq "[a, b]"
}
```

Mit dem Schlüsselwort `vararg` können Sie eine Funktion definieren, die eine beliebige Anzahl von Argumenten annimmt, genau wie `listOf()`. `vararg` ist die Abkürzung für *variable Argumentliste*:

```
// Varargs/VariableArgList.kt
package varargs

fun v(s: String, vararg d: Double) {}

fun main() {
  v("abc", 1.0, 2.0)
  v("def", 1.0, 2.0, 3.0, 4.0)
  v("ghi", 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
}
```

Eine Funktionsdefinition kann nur einen Parameter als `vararg` angeben. Obwohl es möglich ist, ein beliebiges Element in der Parameterliste als `vararg` anzugeben, ist es normalerweise am einfachsten, dies für das letzte zu tun.

`vararg` ermöglicht es Ihnen, eine beliebige Anzahl (einschließlich null) von Argumenten zu übergeben. Alle Argumente müssen vom angegebenen Typ sein. Auf `vararg`-Argumente wird mit dem Parameternamen zugegriffen, der zu einem `Array` wird:

```
// Varargs/VarargSum.kt
package varargs
import atomictest.eq

fun sum(vararg numbers: Int): Int {
  var total = 0
  for (n in numbers) {
    total += n
  }
  return total
}

fun main() {
  sum(13, 27, 44) eq 84
  sum(1, 3, 5, 7, 9, 11) eq 36
  sum() eq 0
}
```

Obwohl `Arrays` und `Lists` ähnlich aussehen, sind sie unterschiedlich implementiert — `List` ist eine reguläre Bibliotheksklasse, während `Array` spezielle Unterstützung auf niedriger Ebene hat. `Array` stammt aus der Anforderung von Kotlin, mit anderen Sprachen, insbesondere Java, kompatibel zu sein.

Im täglichen Programmieren verwenden Sie eine `List`, wenn Sie eine einfache Sequenz benötigen. Verwenden Sie `Arrays` nur, wenn eine Drittanbieter-API ein `Array` erfordert oder wenn Sie mit `varargs` arbeiten.

In den meisten Fällen können Sie einfach ignorieren, dass `vararg` ein `Array` erzeugt, und es behandeln, als ob es eine `List` wäre:

```
// Varargs/VarargLikeList.kt
package varargs
import atomictest.eq

fun evaluate(vararg ints: Int) =
  "Size: ${ints.size}\n" +
  "Sum: ${ints.sum()}\n" +
  "Average: ${ints.average()}"

fun main() {
  evaluate(10, -3, 8, 1, 9) eq """
    Size: 5
```

```
    Sum: 25
    Average: 5.0
  """
}
```

Sie können ein `Array` von Elementen überall dort übergeben, wo ein `vararg` akzeptiert wird. Um ein `Array` zu erstellen, verwenden Sie `arrayOf()` auf die gleiche Weise wie `listOf()`. Ein `Array` ist immer veränderbar. Um ein `Array` in eine Folge von Argumenten (nicht nur ein einzelnes Element des Typs `Array`) zu konvertieren, verwenden Sie den *Spread-Operator*, `*`:

```
// Varargs/SpreadOperator.kt
import varargs.sum
import atomictest.eq

fun main() {
  val array = intArrayOf(4, 5)
  sum(1, 2, 3, *array, 6) eq 21  // [1]
  // Doesn't compile:
  // sum(1, 2, 3, array, 6)

  val list = listOf(9, 10, 11)
  sum(*list.toIntArray()) eq 30  // [2]
}
```

Wenn Sie ein `Array` von primitiven Typen (wie `Int`, `Double` oder `Boolean`) wie im obigen Beispiel übergeben, muss die `Array`-Erstellungsfunktion spezifisch typisiert sein. Wenn Sie `arrayOf(4, 5)` anstelle von `intArrayOf(4, 5)` verwenden, wird Zeile `[1]` einen Fehler erzeugen, der besagt, dass *der abgeleitete Typ ist Array<Int>, aber IntArray wurde erwartet.*

Der Streuoperator funktioniert nur mit Arrays. Wenn Sie eine `List` haben, die Sie als Folge von Argumenten übergeben möchten, konvertieren Sie sie zuerst in ein `Array` und wenden Sie dann den Streuoperator an, wie in **[2]**. Da das Ergebnis ein `Array` eines primitiven Typs ist, müssen wir erneut die spezifische Konvertierungsfunktion `toIntArray()` verwenden.

Der Streuoperator ist besonders hilfreich, wenn Sie `vararg`-Argumente an eine andere Funktion übergeben müssen, die ebenfalls `varargs` erwartet:

```
// Varargs/TwoFunctionsWithVarargs.kt
package varargs
import atomictest.eq

fun first(vararg numbers: Int): String {
  var result = ""
  for (i in numbers) {
    result += "[$i]"
  }
  return result
}

fun second(vararg numbers: Int) =
  first(*numbers)

fun main() {
  second(7, 9, 32) eq "[7][9][32]"
}
```

Kommandozeilenargumente

Beim Aufrufen eines Programms auf der Kommandozeile können Sie ihm eine variable Anzahl von Argumenten übergeben. Um Kommandozeilenargumente zu erfassen, müssen Sie `main()` einen bestimmten Parameter bereitstellen:

```
// Varargs/MainArgs.kt

fun main(args: Array<String>) {
  for (a in args) {
    println(a)
  }
}
```

Der Parameter wird traditionell `args` genannt (obwohl Sie ihn beliebig nennen können), und der Typ für `args` kann nur `Array<String>` (`Array` von `String`) sein.

Wenn Sie IntelliJ IDEA verwenden, können Sie Programmargumente über die Bearbeitung der entsprechenden "Run-Konfiguration" übergeben, wie im letzten Übungsteil für dieses Atom gezeigt.

Sie können auch den `kotlinc`-Compiler verwenden, um ein Befehlszeilenprogramm zu erstellen. Wenn `kotlinc` nicht auf Ihrem Computer vorhanden ist, folgen Sie den Anweisungen auf der Kotlin-Hauptseite[27]. Nachdem Sie den Code für `MainArgs.kt` eingegeben und gespeichert haben, geben Sie Folgendes an einer Eingabeaufforderung ein:

```
kotlinc MainArgs.kt
```

Sie geben die command-line arguments nach dem program invocation ein, so:

```
kotlin MainArgsKt hamster 42 3.14159
```

Sie werden diese Ausgabe sehen:

```
hamster
42
3.14159
```

Wenn Sie einen `String`-Parameter in einen spezifischen Typ umwandeln möchten, stellt Kotlin Konvertierungsfunktionen bereit, wie zum Beispiel `toInt()` für die Umwandlung in einen `Int` und `toFloat()` für die Umwandlung in einen `Float`. Bei der Verwendung dieser Funktionen wird angenommen, dass die Kommandozeilenargumente in einer bestimmten Reihenfolge erscheinen. Hier erwartet das Programm einen `String`, gefolgt von etwas, das in einen `Int` umwandelbar ist, gefolgt von etwas, das in einen `Float` umwandelbar ist:

```
// Varargs/MainArgConversion.kt

fun main(args: Array<String>) {
  if (args.size < 3) return
  val first = args[0]
  val second = args[1].toInt()
  val third = args[2].toFloat()
  println("$first  $second  $third")
}
```

Die erste Zeile in `main()` beendet das Programm, wenn nicht genügend Argumente vorhanden sind. Wenn Sie nichts angeben, das in ein `Int` und ein `Float` umgewandelt werden kann, als zweites und drittes Kommandozeilenargument, werden Sie Laufzeitfehler sehen (versuchen Sie es, um die Fehler zu sehen).

[27] https://kotlinlang.org/

Kompilieren und führen Sie `MainArgConversion.kt` mit denselben Kommandozeilenargumenten aus, die wir zuvor verwendet haben, und Sie werden sehen:

```
hamster 42 3.14159
```

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Mengen

Ein `Set` ist eine Sammlung, die nur ein Element jedes Wertes zulässt.

Die häufigste `Set`-Aktivität ist der Test auf Mitgliedschaft mit `in` oder `contains()`:

```
// Sets/Sets.kt
import atomictest.eq

fun main() {
  val intSet = setOf(1, 1, 2, 3, 9, 9, 4)
  // No duplicates:
  intSet eq setOf(1, 2, 3, 4, 9)

  // Element order is unimportant:
  setOf(1, 2) eq setOf(2, 1)

  // Set membership:
  (9 in intSet) eq true
  (99 in intSet) eq false

  intSet.contains(9) eq true
  intSet.contains(99) eq false

  // Does this set contain another set?
  intSet.containsAll(setOf(1, 9, 2)) eq true

  // Set union:
  intSet.union(setOf(3, 4, 5, 6)) eq
    setOf(1, 2, 3, 4, 5, 6, 9)

  // Set intersection:
  intSet intersect setOf(0, 1, 2, 7, 8) eq
    setOf(1, 2)

  // Set difference:
  intSet subtract setOf(0, 1, 9, 10) eq
```

```
    setOf(2, 3, 4)
  intSet - setOf(0, 1, 9, 10) eq
    setOf(2, 3, 4)
}
```

Dieses Beispiel zeigt:

1. Das Platzieren von doppelten Elementen in einem `Set` entfernt diese Duplikate automatisch.
2. Die Reihenfolge der Elemente ist bei Mengen nicht wichtig. Zwei Mengen sind gleich, wenn sie die gleichen Elemente enthalten.
3. Sowohl `in` als auch `contains()` testen auf Mitgliedschaft.
4. Sie können die üblichen Venn-Diagramm-Operationen wie Überprüfung auf Teilmengen, Vereinigung, Schnittmenge und Differenz durchführen, entweder mit Punktnotation (`set.union(other)`) oder Infix-Notation (`set intersect other`). Die Funktionen `union`, `intersect` und `subtract` können mit Infix-Notation verwendet werden.
5. Die Mengendifferenz kann entweder mit `subtract()` oder dem Minus-Operator ausgedrückt werden.

Um Duplikate aus einer `Liste` zu entfernen, konvertieren Sie sie in ein `Set`:

```
// Sets/RemoveDuplicates.kt
import atomictest.eq

fun main() {
  val list = listOf(3, 3, 2, 1, 2)
  list.toSet() eq setOf(1, 2, 3)
  list.distinct() eq listOf(3, 2, 1)
  "abbcc".toSet() eq setOf('a', 'b', 'c')
}
```

Sie können auch `distinct()` verwenden, das eine `List` zurückgibt. Sie können `toSet()` auf einem `String` aufrufen, um ihn in eine Menge einzigartiger Zeichen umzuwandeln.

Wie bei `List` bietet Kotlin zwei Erstellungsfunktionen für `Set`. Das Ergebnis von `setOf()` ist schreibgeschützt. Um ein veränderbares `Set` zu erstellen, verwenden Sie `mutableSetOf()`:

```
// Sets/MutableSet.kt
import atomictest.eq

fun main() {
  val mutableSet = mutableSetOf<Int>()
  mutableSet += 42
  mutableSet += 42
  mutableSet eq setOf(42)
  mutableSet -= 42
  mutableSet eq setOf<Int>()
}
```

Die Operatoren += und -= fügen Elemente zu `Sets` hinzu bzw. entfernen sie, genau wie bei `Lists`.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Karten

> Eine `Map` verbindet *keys* mit *values* und sucht einen Wert anhand eines Schlüssels.

Sie erstellen eine `Map`, indem Sie `mapOf()` Schlüssel-Wert-Paare bereitstellen. Mit `to` trennen wir jeden Schlüssel von seinem zugehörigen Wert:

```
// Maps/Maps.kt
import atomictest.eq

fun main() {
  val constants = mapOf(
    "Pi" to 3.141,
    "e" to 2.718,
    "phi" to 1.618
  )
  constants eq
    "{Pi=3.141, e=2.718, phi=1.618}"

  // Look up a value from a key:
  constants["e"] eq 2.718              // [1]
  constants.keys eq setOf("Pi", "e", "phi")
  constants.values eq "[3.141, 2.718, 1.618]"

  var s = ""
  // Iterate through key-value pairs:
  for (entry in constants) {           // [2]
    s += "${entry.key}=${entry.value}, "
  }
  s eq "Pi=3.141, e=2.718, phi=1.618,"

  s = ""
  // Unpack during iteration:
  for ((key, value) in constants)      // [3]
    s += "$key=$value, "
  s eq "Pi=3.141, e=2.718, phi=1.618,"
}
```

- **[1]** Der `[]` Operator sucht einen Wert mithilfe eines Schlüssels. Sie können alle Schlüssel mit `keys` und alle Werte mit `values` erzeugen. Der Aufruf von `keys` erzeugt eine `Set`, da alle Schlüssel in einem `Map` einzigartig sein müssen, andernfalls gäbe es eine Mehrdeutigkeit bei einer Suche.
- **[2]** Das Iterieren durch ein `Map` erzeugt Schlüssel-Wert-Paare als Mapeinträge.
- **[3]** Sie können Schlüssel und Werte beim Iterieren entpacken.

Ein einfaches `Map` ist schreibgeschützt. Hier ist ein `MutableMap`:

```
// Maps/MutableMaps.kt
import atomictest.eq

fun main() {
  val m =
    mutableMapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  m[5] = "5ive"
  m[5] eq "5ive"
  m += 4 to "four"
  m eq mapOf(5 to "5ive",
    4 to "four", 6 to "six")
}
```

`map[key] = value` fügt den `Wert` hinzu oder ändert ihn, der mit dem `Schlüssel` verknüpft ist. Sie können auch explizit ein Paar hinzufügen, indem Sie `map += key to value` verwenden.

`mapOf()` und `mutableMapOf()` bewahren die Reihenfolge, in der die Elemente in die `Map` eingefügt werden. Dies ist nicht für andere Typen von `Map` garantiert.

Eine schreibgeschützte `Map` erlaubt keine Änderungen:

```
// Maps/ReadOnlyMaps.kt
import atomictest.eq

fun main() {
  val m = mapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  // m[5] = "5ive" // Fails
  // m += (4 to "four") // Fails
  m + (4 to "four") // Doesn't change m
  m eq mapOf(5 to "five", 6 to "six")
  val m2 = m + (4 to "four")
  m2 eq mapOf(
    5 to "five", 6 to "six", 4 to "four")
}
```

Die Definition von `m` erstellt eine `Map`, die `Int`s mit `String`s verknüpft. Wenn wir versuchen, einen `String` zu ersetzen, gibt Kotlin einen Fehler aus.

Ein Ausdruck mit `+` erstellt eine neue `Map`, die sowohl die alten Elemente als auch das neue enthält, aber die ursprüngliche `Map` nicht beeinflusst. Die einzige Möglichkeit, ein Element zu einer unveränderlichen `Map` "hinzuzufügen", besteht darin, eine neue `Map` zu erstellen.

Eine `Map` gibt `null` zurück, wenn sie keinen Eintrag für einen gegebenen Schlüssel enthält. Wenn Sie ein Ergebnis benötigen, das nicht `null` sein kann, verwenden Sie `getValue()` und fangen Sie `NoSuchElementException` ab, falls der Schlüssel fehlt:

```
// Maps/GetValue.kt
import atomictest.*

fun main() {
  val map = mapOf('a' to "attempt")
  map['b'] eq null
  capture {
    map.getValue('b')
  } eq "NoSuchElementException: " +
    "Key b is missing in the map."
  map.getOrDefault('a', "??") eq "attempt"
  map.getOrDefault('b', "??") eq "??"
}
```

`getOrDefault()` ist normalerweise eine angenehmere Alternative zu `null` oder einer Ausnahme.

Sie können Klasseninstanzen als Werte in einem `Map` speichern. Hier ist ein Map, das einen `Contact` anhand eines Zahlen-`String` abruft:

```
// Maps/ContactMap.kt
package maps
import atomictest.eq

class Contact(
  val name: String,
  val phone: String
) {
  override fun toString() =
    "Contact('$name', '$phone')"
}

fun main() {
  val miffy = Contact("Miffy", "1-234-567890")
  val cleo = Contact("Cleo", "098-765-4321")
  val contacts = mapOf(
    miffy.phone to miffy,
    cleo.phone to cleo)
  contacts["1-234-567890"] eq miffy
  contacts["1-111-111111"] eq null
}
```

Es ist möglich, Klasseninstanzen als Schlüssel in einer `Map` zu verwenden, aber das ist komplizierter, daher besprechen wir es später im Buch.

- -

`Maps` sehen aus wie einfache kleine Datenbanken. Sie werden manchmal *assoziative Arrays* genannt, weil sie Schlüssel mit Werten verknüpfen. Obwohl sie im Vergleich zu einer voll ausgestatteten Datenbank ziemlich begrenzt sind, sind sie dennoch bemerkenswert nützlich (und weitaus effizienter als eine Datenbank).

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Eigenschaftszugriffe

> Um eine Eigenschaft zu lesen, verwenden Sie ihren Namen. Um einer veränderlichen Eigenschaft einen Wert zuzuweisen, verwenden Sie den Zuweisungsoperator =.

Dies liest und schreibt die Eigenschaft `i`:

```
// PropertyAccessors/Data.kt
package propertyaccessors
import atomictest.eq

class Data(var i: Int)

fun main() {
  val data = Data(10)
  data.i eq 10 // Read the 'i' property
  data.i = 20  // Write to the 'i' property
}
```

Dies scheint ein direkter Zugriff auf das Speicherelement namens `i` zu sein. Allerdings ruft Kotlin Funktionen auf, um die Lese- und Schreiboperationen durchzuführen. Wie erwartet, lesen und schreiben diese Funktionen standardmäßig die in `i` gespeicherten Daten. In diesem Abschnitt lernen Sie, Ihre eigenen *Eigenschaftszugriffe* zu schreiben, um die Lese- und Schreibaktionen anzupassen.

Der Zugriff, der verwendet wird, um den Wert einer Eigenschaft zu erhalten, wird *Getter* genannt. Sie erstellen einen Getter, indem Sie `get()` direkt nach der Eigenschaftsdefinition definieren. Der Zugriff, der verwendet wird, um eine änderbare Eigenschaft zu modifizieren, wird *Setter* genannt. Sie erstellen einen Setter, indem Sie `set()` direkt nach der Eigenschaftsdefinition definieren.

Die in dem folgenden Beispiel definierten Eigenschaftszugriffe imitieren die von Kotlin generierten Standardimplementierungen. Wir zeigen zusätzliche Informationen an, damit Sie sehen können, dass die Eigenschaftszugriffe tatsächlich während der

Lese- und Schreibvorgänge aufgerufen werden. Wir rücken `get()` und `set()` ein, um sie visuell mit der Eigenschaft zu verknüpfen, aber die eigentliche Verknüpfung erfolgt, weil `get()` und `set()` direkt nach dieser Eigenschaft definiert sind (Kotlin kümmert sich nicht um die Einrückung):

```
// PropertyAccessors/Default.kt
package propertyaccessors
import atomictest.*

class Default {
  var i: Int = 0
    get() {
      trace("get()")
      return field       // [1]
    }
    set(value) {
      trace("set($value)")
      field = value      // [2]
    }
}

fun main() {
  val d = Default()
  d.i = 2
  trace(d.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

Die Reihenfolge der Definition von `get()` und `set()` ist unwichtig. Sie können `get()` definieren, ohne `set()` zu definieren, und umgekehrt.

Das Standardverhalten einer Eigenschaft gibt ihren gespeicherten Wert über einen Getter zurück und modifiziert ihn mit einem Setter—die Aktionen von **[1]** und **[2]**. Innerhalb des Getters und Setters wird der gespeicherte Wert indirekt mit dem Schlüsselwort `field` manipuliert, das nur innerhalb dieser beiden Funktionen zugänglich ist.

Das nächste Beispiel verwendet die Standardimplementierung des Getters und fügt einen Setter hinzu, um Änderungen an der Eigenschaft `n` nachzuverfolgen:

```
// PropertyAccessors/LogChanges.kt
package propertyaccessors
import atomictest.*

class LogChanges {
  var n: Int = 0
    set(value) {
      trace("$field becomes $value")
      field = value
    }
}

fun main() {
  val lc = LogChanges()
  lc.n eq 0
  lc.n = 2
  lc.n eq 2
  trace eq "0 becomes 2"
}
```

Wenn Sie eine Eigenschaft als `private` definieren, werden beide Zugriffsmethoden `private`. Sie können auch den Setter `private` machen und den Getter `public`. Dann können Sie die Eigenschaft außerhalb der Klasse lesen, aber ihren Wert nur innerhalb der Klasse ändern:

```
// PropertyAccessors/Counter.kt
package propertyaccessors
import atomictest.eq

class Counter {
  var value: Int = 0
    private set
  fun inc() = value++
}

fun main() {
  val counter = Counter()
  repeat(10) {
    counter.inc()
```

```
  }
  counter.value eq 10
}
```

Mit `private set` kontrollieren wir die Eigenschaft `value`, sodass sie nur um eins erhöht werden kann.

Normale Eigenschaften speichern ihre Daten in einem Feld. Man kann auch eine Eigenschaft erstellen, die kein Feld hat:

```
// PropertyAccessors/Hamsters.kt
package propertyaccessors
import atomictest.eq

class Hamster(val name: String)

class Cage(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  val capacity: Int
    get() = maxCapacity - hamsters.size
  val full: Boolean
    get() = hamsters.size == maxCapacity
  fun put(hamster: Hamster): Boolean =
    if (full)
      false
    else {
      hamsters += hamster
      true
    }
  fun take(): Hamster =
    hamsters.removeAt(0)
}

fun main() {
  val cage = Cage(2)
  cage.full eq false
  cage.capacity eq 2
  cage.put(Hamster("Alice")) eq true
  cage.put(Hamster("Bob")) eq true
  cage.full eq true
  cage.capacity eq 0
```

```
  cage.put(Hamster("Charlie")) eq false
  cage.take()
  cage.capacity eq 1
}
```

Die Eigenschaften `capacity` und `full` enthalten keinen zugrunde liegenden Zustand—sie werden zum Zeitpunkt jedes Zugriffs berechnet. Sowohl `capacity` als auch `full` sind ähnlich wie Funktionen, und Sie können sie als solche definieren:

```
// PropertyAccessors/Hamsters2.kt
package propertyaccessors

class Cage2(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  fun capacity(): Int =
    maxCapacity - hamsters.size
  fun isFull(): Boolean =
    hamsters.size == maxCapacity
}
```

In diesem Fall verbessert die Verwendung von Eigenschaften die Lesbarkeit, da Kapazität und Fülle Eigenschaften des Käfigs sind. Wandeln Sie jedoch nicht einfach alle Ihre Funktionen in Eigenschaften um—sehen Sie sich zuerst an, wie sie sich lesen.

- -

Die Kotlin-Stilrichtlinie bevorzugt Eigenschaften gegenüber Funktionen, wenn der Wert günstig zu berechnen ist und die Eigenschaft bei jedem Aufruf dasselbe Ergebnis liefert, solange sich der Objektzustand nicht geändert hat.

Eigenschaftszugriffe bieten eine Art Schutz für Eigenschaften. Viele objektorientierte Sprachen verlassen sich darauf, ein physisches Feld `private` zu machen, um den Zugriff auf diese Eigenschaft zu kontrollieren. Mit Eigenschaftszugriffen können Sie Code hinzufügen, um diesen Zugriff zu kontrollieren oder zu verändern, während Sie jedem erlauben, eine Eigenschaft zu verwenden.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Zusammenfassung 2

> Dieses Atom fasst die Atome in Abschnitt II zusammen und überprüft sie, von Objects Everywhere bis Property Accessors.

Wenn Sie ein erfahrener Programmierer sind, ist dies Ihr nächstes Atom nach Summary 1, und Sie werden die Atome danach der Reihe nach durchgehen.

Neue Programmierer sollten dieses Atom lesen und die Übungen zur Überprüfung durchführen. Wenn Ihnen hier Informationen unklar sind, gehen Sie zurück und studieren Sie das Atom zu diesem Thema.

Die Themen erscheinen in einer geeigneten Reihenfolge für erfahrene Programmierer, was nicht der gleichen Reihenfolge der Atome im Buch entspricht. Zum Beispiel beginnen wir mit der Einführung von Paketen und Importen, damit wir unser minimales Test-Framework für den Rest des Atoms verwenden können.

Pakete & Testen

Eine beliebige Anzahl von wiederverwendbaren Bibliothekskomponenten kann unter einem einzigen Bibliotheksnamen mit dem `package`-Schlüsselwort gebündelt werden:

```
// Summary2/ALibrary.kt
package com.yoururl.libraryname

// Components to reuse ...
fun f() = "result"
```

Sie können mehrere Komponenten in einer einzigen Datei platzieren oder Komponenten auf mehrere Dateien mit demselben Paketnamen verteilen. Hier haben wir `f()` als einzige Komponente definiert.

Um es eindeutig zu machen, beginnt der Paketname konventionell mit Ihrem umgekehrten Domain-Namen. In diesem Beispiel ist der Domain-Name `yoururl.com`.

In Kotlin kann der Paketname unabhängig von dem Verzeichnis sein, in dem sich seine Inhalte befinden. Java verlangt, dass die Verzeichnisstruktur mit dem vollqualifizierten Paketnamen übereinstimmt, sodass das Paket `com.yoururl.libraryname` im Verzeichnis `com/yoururl/libraryname` liegen sollte. Für gemischte Kotlin- und Java-Projekte empfiehlt der Kotlin-Stilführer dieselbe Praxis. Für reine Kotlin-Projekte platzieren Sie das Verzeichnis `libraryname` auf der obersten Ebene der Verzeichnisstruktur Ihres Projekts.

Eine `Importanweisung` bringt einen oder mehrere Namen in den aktuellen Namensraum:

```
// Summary2/UseALibrary.kt
import com.yoururl.libraryname.*

fun main() {
  val x = f()
}
```

Der Stern nach `libraryname` weist Kotlin an, alle Komponenten einer Bibliothek zu importieren. Sie können auch Komponenten einzeln auswählen; Details finden Sie unter Pakete.

Im restlichen Teil dieses Buches verwenden wir `package`-Anweisungen für jede Datei, die Funktionen, Klassen usw. außerhalb von `main()` definiert. Dies verhindert Namenskonflikte mit anderen Dateien im Buch. In der Regel setzen wir keine `package`-Anweisung in eine Datei, die *nur* ein `main()` enthält.

Eine wichtige Bibliothek für dieses Buch ist `atomictest`, unser einfaches Test-Framework. `atomictest` ist definiert in Anhang A: AtomicTest, obwohl es Sprachmerkmale verwendet, die Sie zu diesem Zeitpunkt im Buch noch nicht verstehen werden.

Nach dem Importieren von `atomictest` verwenden Sie `eq` (gleich) und `neq` (ungleich) fast so, als wären sie Sprachschlüsselwörter:

```
// Summary2/UsingAtomicTest.kt
import atomictest.*

fun main() {
  val pi = 3.14
  val pie = "A round dessert"
  pi eq 3.14
  pie eq "A round dessert"
  pi neq pie
}
/* Output:
3.14
A round dessert
3.14
*/
```

Die Fähigkeit, eq/neq ohne Punkte oder Klammern zu verwenden, wird als *Infix-Notation* bezeichnet. Sie können `infix`-Funktionen entweder auf die reguläre Weise aufrufen: `pi.eq(3.14)`, oder unter Verwendung der Infix-Notation: `pi eq 3.14`. Sowohl eq als auch neq sind Wahrheitsaussagen, die das Ergebnis von der linken Seite der eq/neq-Anweisung anzeigen, sowie eine Fehlermeldung, wenn der Ausdruck auf der rechten Seite von eq nicht gleichwertig zur linken ist (oder *gleichwertig* ist, im Fall von neq). Auf diese Weise sehen Sie verifizierte Ergebnisse im Quellcode.

`atomictest.trace` verwendet die Funktionsaufruf-Syntax, um Ergebnisse hinzuzufügen, die dann mit eq validiert werden können:

```
// Testing/UsingTrace.kt
import atomictest.*

fun main() {
  trace("Hello,")
  trace(47)
  trace("World!")
  trace eq """
    Hello,
    47
    World!
  """
}
```

Sie können `println()` effektiv durch `trace()` ersetzen.

Objekte überall

Kotlin ist eine *hybrid objekt-funktionale* Sprache: Sie unterstützt sowohl objektorientierte als auch funktionale Programmierparadigmen.

Objekte enthalten `val`s und `var`s, um Daten zu speichern (diese werden *Eigenschaften* genannt) und führen Operationen mit Funktionen aus, die innerhalb einer Klasse definiert sind, sogenannte *Mitgliedsfunktionen* (wenn es eindeutig ist, sagen wir einfach "Funktionen"). Eine *Klasse* definiert Eigenschaften und Mitgliedsfunktionen für das, was im Wesentlichen ein neuer, benutzerdefinierter Datentyp ist. Wenn Sie ein `val` oder `var` einer Klasse erstellen, nennt man dies *ein Objekt erstellen* oder *eine Instanz erstellen.*

Eine besonders nützliche Art von Objekt ist der *Container*, auch *Sammlung* genannt. Ein Container ist ein Objekt, das andere Objekte hält. In diesem Buch verwenden wir oft die `List`, da sie die vielseitigste Sequenz ist. Hier führen wir mehrere Operationen an einer `List` durch, die `Doubles` enthält. `listOf()` erstellt eine neue `List` aus ihren Argumenten:

```
// Summary2/ListCollection.kt
import atomictest.eq

fun main() {
  val lst = listOf(19.2, 88.3, 22.1)
  lst[1] eq 88.3  // Indexing
  lst.reversed() eq listOf(22.1, 88.3, 19.2)
  lst.sorted() eq listOf(19.2, 22.1, 88.3)
  lst.sum() eq 129.6
}
```

Kein `import`-Statement ist erforderlich, um eine `List` zu verwenden.

Kotlin verwendet eckige Klammern für die Indexierung in Sequenzen. Die Indexierung beginnt bei Null.

Dieses Beispiel zeigt auch einige der vielen Standardbibliotheksfunktionen, die für `Lists` verfügbar sind: `sorted()`, `reversed()`, und `sum()`. Um diese Funktionen zu verstehen, konsultieren Sie die Kotlin Dokumentation[28] online.

[28] https://kotlinlang.org/docs/reference/

Wenn Sie `sorted()` oder `reversed()` aufrufen, wird `lst` nicht verändert. Stattdessen wird eine neue `List` erstellt und zurückgegeben, die das gewünschte Ergebnis enthält. Dieser Ansatz, das Originalobjekt niemals zu verändern, ist durchgehend in den Kotlin-Bibliotheken konsistent, und Sie sollten bestrebt sein, diesem Muster zu folgen, wenn Sie Ihren eigenen Code schreiben.

Klassen erstellen

Eine Klassendefinition besteht aus dem Schlüsselwort `class`, einem Namen für die Klasse und einem optionalen Körper. Der Körper enthält Eigenschaftsdefinitionen (`vals` und `vars`) und Funktionsdefinitionen.

Dieses Beispiel definiert eine `NoBody`-Klasse ohne Körper und Klassen mit `val`-Eigenschaften:

```
// Summary2/ClassBodies.kt
package summary2

class NoBody

class SomeBody {
  val name = "Janet Doe"
}

class EveryBody {
  val all = listOf(SomeBody(),
    SomeBody(), SomeBody())
}

fun main() {
  val nb = NoBody()
  val sb = SomeBody()
  val eb = EveryBody()
}
```

Um eine Instanz einer Klasse zu erstellen, setzen Sie Klammern nach ihrem Namen, sowie Argumente, falls diese erforderlich sind.

Eigenschaften innerhalb von Klassenkörpern können jeden Typ haben. `SomeBody` enthält eine Eigenschaft vom Typ `String`, und die Eigenschaft von `EveryBody` ist eine `List`, die `SomeBody`-Objekte hält.

Hier ist eine Klasse mit Mitgliedsfunktionen:

```
// Summary2/Temperature.kt
package summary2
import atomictest.eq

class Temperature {
  var current = 0.0
  var scale = "f"
  fun setFahrenheit(now: Double) {
    current = now
    scale = "f"
  }
  fun setCelsius(now: Double) {
    current = now
    scale = "c"
  }
  fun getFahrenheit(): Double =
    if (scale == "f")
      current
    else
      current * 9.0 / 5.0 + 32.0
  fun getCelsius(): Double =
    if (scale == "c")
      current
    else
      (current - 32.0) * 5.0 / 9.0
}

fun main() {
  val temp = Temperature()   // [1]
  temp.setFahrenheit(98.6)
  temp.getFahrenheit() eq 98.6
  temp.getCelsius() eq 37.0
  temp.setCelsius(100.0)
  temp.getFahrenheit() eq 212.0
}
```

Diese Mitgliedsfunktionen sind genau wie die auf oberster Ebene definierten Funktionen *außerhalb* von Klassen, außer dass sie zur Klasse gehören und uneingeschränkten Zugriff auf die anderen Mitglieder der Klasse haben, wie `current` und `scale`.

Mitgliedsfunktionen können auch andere Mitgliedsfunktionen in derselben Klasse ohne Qualifikation aufrufen.

- **[1]** Obwohl `temp` ein `val` ist, modifizieren wir später das `Temperature`-Objekt. Die `val`-Definition verhindert, dass die Referenz `temp` auf ein neues Objekt umgeschrieben wird, schränkt jedoch das Verhalten des Objekts selbst nicht ein.

Die folgenden zwei Klassen sind die Basis eines Tic-Tac-Toe-Spiels:

```
// Summary2/TicTacToe.kt
package summary2
import atomictest.eq

class Cell {
  var entry = ' '                    // [1]
  fun setValue(e: Char): String =    // [2]
    if (entry == ' ' &&
      (e == 'X' || e == 'O')) {
      entry = e
      "Successful move"
    } else
      "Invalid move"
}

class Grid {
  val cells = listOf(
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell())
  )
  fun play(e: Char, x: Int, y: Int): String =
    if (x !in 0..2 || y !in 0..2)
      "Invalid move"
    else
      cells[x][y].setValue(e)        // [3]
}

fun main() {
  val grid = Grid()
  grid.play('X', 1, 1) eq "Successful move"
  grid.play('X', 1, 1) eq "Invalid move"
```

```
  grid.play('O', 1, 3) eq "Invalid move"
}
```

Die `Grid`-Klasse enthält eine `List`, die drei `Lists` enthält, von denen jede drei `Cells` enthält — eine Matrix.

- **[1]** Die `entry`-Eigenschaft in `Cell` ist ein `var`, sodass sie modifiziert werden kann. Die einfachen Anführungszeichen in der Initialisierung erzeugen einen `Char`-Typ, daher müssen alle Zuweisungen zu `entry` ebenfalls `Chars` sein.
- **[2]** `setValue()` prüft, ob die `Cell` verfügbar ist und ob Sie das richtige Zeichen übergeben haben. Es gibt ein `String`-Ergebnis zurück, um Erfolg oder Misserfolg anzuzeigen.
- **[3]** `play()` überprüft, ob die `x`- und `y`-Argumente im Bereich liegen, und indiziert dann in die Matrix, wobei es sich auf die von `setValue()` durchgeführten Tests stützt.

Konstruktoren

Konstruktoren erstellen neue Objekte. Sie übergeben Informationen an einen Konstruktor mithilfe seiner Parameterliste, die direkt nach dem Klassennamen in Klammern gesetzt wird. Ein Konstruktoraufruf sieht daher wie ein Funktionsaufruf aus, außer dass der Anfangsbuchstabe des Namens großgeschrieben wird (gemäß dem Kotlin-Stilguide). Der Konstruktor gibt ein Objekt der Klasse zurück:

```
// Summary2/WildAnimals.kt
package summary2
import atomictest.eq

class Badger(id: String, years: Int) {
  val name = id
  val age = years
  override fun toString() =
    "Badger: $name, age: $age"
}

class Snake(
  var type: String,
  var length: Double
```

```
) {
  override fun toString() =
    "Snake: $type, length: $length"
}

class Moose(
  val age: Int,
  val height: Double
) {
  override fun toString() =
    "Moose, age: $age, height: $height"
}

fun main() {
  Badger("Bob", 11) eq "Badger: Bob, age: 11"
  Snake("Garden", 2.4) eq
    "Snake: Garden, length: 2.4"
  Moose(16, 7.2) eq
    "Moose, age: 16, height: 7.2"
}
```

Die Parameter `id` und `years` in `Badger` sind nur im *Konstruktor-Körper* verfügbar. Der Konstruktor-Körper besteht aus den Codezeilen, die keine Funktionsdefinitionen sind; in diesem Fall die Definitionen für `name` und `age`.

Oft möchte man, dass die Konstruktor-Parameter in Teilen der Klasse verfügbar sind, die nicht zum Konstruktor-Körper gehören, ohne dass man neue Bezeichner explizit definieren muss, wie wir es bei `name` und `age` getan haben. Wenn Sie Ihre Parameter als `vars` oder `vals` definieren, werden sie zu Eigenschaften und sind überall in der Klasse zugänglich. Sowohl `Snake` als auch `Moose` verwenden diesen Ansatz, und Sie können sehen, dass die Konstruktor-Parameter jetzt innerhalb ihrer jeweiligen `toString()`-Funktionen verfügbar sind.

Mit `val` deklarierte Konstruktor-Parameter können nicht geändert werden, aber die mit `var` deklarierten schon.

Wann immer Sie ein Objekt in einer Situation verwenden, die einen `String` erwartet, erzeugt Kotlin eine `String`-Darstellung dieses Objekts, indem es seine `toString()`-Mitgliedsfunktion aufruft. Um eine `toString()` zu definieren, müssen Sie ein neues Schlüsselwort verstehen: `override`. Dies ist notwendig (Kotlin besteht darauf), weil `toString()` bereits definiert ist. `override` teilt Kotlin mit, dass wir tatsächlich die

Standard-`toString()` durch unsere eigene Definition ersetzen wollen. Die Explizität von `override` macht dies dem Leser klar und hilft, Fehler zu vermeiden.

Beachten Sie das Format der mehrzeiligen Parameterliste für `Snake` und `Moose` — dies ist der empfohlene Standard, wenn Sie zu viele Parameter haben, um sie in eine Zeile zu passen, sowohl für Konstruktoren als auch für Funktionen.

Einschränkung der Sichtbarkeit

Kotlin bietet *Zugriffsmodifikatoren*, die denen in anderen Sprachen wie C++ oder Java ähnlich sind. Diese ermöglichen es den Erstellern von Komponenten, zu entscheiden, was für den Client-Programmierer verfügbar ist. Zu den Zugriffsmodifikatoren von Kotlin gehören die Schlüsselwörter `public`, `private`, `protected` und `internal`. `protected` wird später erklärt.

Ein Zugriffsmodifikator wie `public` oder `private` erscheint vor der Definition einer Klasse, Funktion oder Eigenschaft. Jeder Zugriffsmodifikator steuert nur den Zugriff auf diese spezifische Definition.

Eine `public`-Definition ist für jeden verfügbar, insbesondere für den Client-Programmierer, der diese Komponente verwendet. Daher wirken sich alle Änderungen an einer `public`-Definition auf den Client-Code aus.

Wenn Sie keinen Modifikator angeben, ist Ihre Definition automatisch `public`. Aus Gründen der Klarheit geben Programmierer in bestimmten Fällen manchmal trotzdem redundant `public` an.

Wenn Sie eine Klasse, eine top-level Funktion oder Eigenschaft als `private` definieren, ist sie nur innerhalb dieser Datei verfügbar:

```
// Summary2/Boxes.kt
package summary2
import atomictest.*

private var count = 0                    // [1]

private class Box(val dimension: Int) { // [2]
  fun volume() =
    dimension * dimension * dimension
  override fun toString() =
    "Box volume: ${volume()}"
}

private fun countBox(box: Box) {         // [3]
  trace("$box")
  count++
}

fun countBoxes() {
  countBox(Box(4))
  countBox(Box(5))
}

fun main() {
  countBoxes()
  trace("$count boxes")
  trace eq """
    Box volume: 64
    Box volume: 125
    2 boxes
  """
}
```

Sie können auf `private` Eigenschaften (**[1]**), Klassen (**[2]**) und Funktionen (**[3]**) nur von anderen Funktionen und Klassen in der Datei `Boxes.kt` zugreifen. Kotlin verhindert, dass Sie auf `private` Top-Level-Elemente von einer anderen Datei aus zugreifen.

Klassenmitglieder können `private` sein:

```
// Summary2/JetPack.kt
package summary2
import atomictest.eq

class JetPack(
  private var fuel: Double   // [1]
) {
  private var warning = false
  private fun burn() =       // [2]
    if (fuel - 1 <= 0) {
      fuel = 0.0
      warning = true
    } else
      fuel -= 1
  public fun fly() = burn()  // [3]
  fun check() =              // [4]
    if (warning)             // [5]
      "Warning"
    else
      "OK"
}

fun main() {
  val jetPack = JetPack(3.0)
  while (jetPack.check() != "Warning") {
    jetPack.check() eq "OK"
    jetPack.fly()
  }
  jetPack.check() eq "Warning"
}
```

- **[1]** `fuel` und `warning` sind beide `private` Eigenschaften und können nicht von Nicht-Mitgliedern von `JetPack` verwendet werden.
- **[2]** `burn()` ist `private` und somit nur innerhalb von `JetPack` zugänglich.
- **[3]** `fly()` und `check()` sind `public` und können überall verwendet werden.
- **[4]** Kein Zugriffsmodifizierer bedeutet `public` Sichtbarkeit.
- **[5]** Nur Mitglieder derselben Klasse können auf `private` Mitglieder zugreifen.

Da eine `private` Definition *nicht* für alle verfügbar ist, kann man sie im Allgemeinen ändern, ohne sich um den Client-Programmierer zu sorgen. Als Bibliotheksdesigner

hält man normalerweise alles so `private` wie möglich und gibt nur Funktionen und Klassen frei, die Client-Programmierer verwenden sollen. Um die Größe und Komplexität der Beispielauflistungen in diesem Buch zu begrenzen, verwenden wir `private` nur in speziellen Fällen.

Jede Funktion, bei der Sie sicher sind, dass es sich nur um eine *Hilfsfunktion* handelt, kann `private` gemacht werden, um sicherzustellen, dass Sie sie nicht versehentlich anderswo verwenden und sich damit verbieten, die Funktion zu ändern oder zu entfernen.

Es kann nützlich sein, große Programme in *Module* zu unterteilen. Ein Modul ist ein logisch unabhängiger Teil einer Codebasis. Eine `internal` Definition ist nur innerhalb des Moduls zugänglich, in dem sie definiert ist. Die Art und Weise, wie Sie ein Projekt in Module unterteilen, hängt vom Build-System ab (wie Gradle[29] oder Maven[30]) und liegt außerhalb des Rahmens dieses Buches.

Module sind ein Konzept auf höherer Ebene, während *Pakete* eine feiner abgestufte Strukturierung ermöglichen.

Ausnahmen

Betrachten Sie `toDouble()`, das einen `String` in ein `Double` umwandelt. Was passiert, wenn Sie es für einen `String` aufrufen, der nicht in ein `Double` übersetzt wird?

```
// Summary2/ToDoubleException.kt

fun main() {
  // val i = "$1.9".toDouble()
}
```

Das Auskommentieren der Zeile in `main()` erzeugt eine Ausnahme. Hier ist die fehlerhafte Zeile auskommentiert, damit der Bau des Buches nicht gestoppt wird (der überprüft, ob jedes Beispiel wie erwartet kompiliert und ausgeführt wird).

Wenn eine Ausnahme ausgelöst wird, stoppt der aktuelle Ausführungspfad, und das Ausnahmeobjekt wird aus dem aktuellen Kontext herausgeschleudert. Wenn eine

[29] https://gradle.org/
[30] https://maven.apache.org/

Ausnahme nicht abgefangen wird, bricht das Programm ab und zeigt einen *Stack-Trace* mit detaillierten Informationen an.

Um das Anzeigen von Ausnahmen durch Kommentieren und Auskommentieren von Code zu vermeiden, speichert `atomictest.capture()` die Ausnahme und vergleicht sie mit dem, was wir erwarten:

```
// Summary2/AtomicTestCapture.kt
import atomictest.*

fun main() {
  capture {
    "$1.9".toDouble()
  } eq "NumberFormatException: " +
    """For input string: "$1.9""""
}
```

`capture()` ist speziell für dieses Buch konzipiert, damit Sie die Ausnahme sehen und wissen, dass die Ausgabe vom Build-System des Buches überprüft wurde.

Eine weitere Strategie, wenn Ihre Funktion das erwartete Ergebnis nicht erfolgreich liefern kann, ist die Rückgabe von `null`. Später in Nullable Types diskutieren wir, wie `null` den Typ des resultierenden Ausdrucks beeinflusst.

Um eine Ausnahme zu werfen, verwenden Sie das Schlüsselwort `throw`, gefolgt von der Ausnahme, die Sie werfen möchten, zusammen mit allen Argumenten, die sie möglicherweise benötigt. `quadraticZeroes()` im folgenden Beispiel löst die quadratische Gleichung[31], die eine Parabel definiert:

$$ax^2 + bx + c = 0$$

Die Lösung ist die *quadratische Formel*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Die quadratische Formel

Das Beispiel findet die *Nullstellen* der Parabel, wo die Linien die x-Achse schneiden. Wir werfen Ausnahmen für zwei Einschränkungen:

[31] https://en.wikipedia.org/wiki/Quadratic_formula

1. `a` darf nicht null sein.
2. Damit Nullstellen existieren, darf $b^2 - 4ac$ nicht negativ sein.

Wenn Nullstellen existieren, gibt es zwei, daher erstellen wir die `Roots`-Klasse, um die Rückgabewerte zu halten:

```
// Summary2/Quadratic.kt
package summary2
import kotlin.math.sqrt
import atomictest.*

class Roots(
  val root1: Double,
  val root2: Double
)

fun quadraticZeroes(
  a: Double,
  b: Double,
  c: Double
): Roots {
  if (a == 0.0)
    throw IllegalArgumentException(
      "a is zero")
  val underRadical = b * b - 4 * a * c
  if (underRadical < 0)
    throw IllegalArgumentException(
      "Negative underRadical: $underRadical")
  val squareRoot = sqrt(underRadical)
  val root1 = (-b - squareRoot) / (2 * a)
  val root2 = (-b + squareRoot) / (2 * a)
  return Roots(root1, root2)
}

fun main() {
  capture {
    quadraticZeroes(0.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
    "a is zero"
  capture {
    quadraticZeroes(3.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
```

```
    "Negative underRadical: -44.0"
  val roots = quadraticZeroes(1.0, 2.0, -8.0)
  roots.root1 eq -4.0
  roots.root2 eq 2.0
}
```

Hier verwenden wir die Standard-Ausnahmeklasse `IllegalArgumentException`. Später werden Sie lernen, Ihre eigenen Ausnahmetypen zu definieren und sie spezifisch an Ihre Umstände anzupassen. Ihr Ziel ist es, die nützlichsten Nachrichten zu generieren, um die Unterstützung Ihrer Anwendung in der Zukunft zu vereinfachen.

Listen

`Lists` sind Kotlins grundlegender sequentieller Containertyp. Sie erstellen eine schreibgeschützte Liste mit `listOf()` und eine veränderbare Liste mit `mutableListOf()`:

```
// Summary2/ReadonlyVsMutableList.kt
import atomictest.*

fun main() {
  val ints = listOf(5, 13, 9)
  // ints.add(11) // 'add()' not available
  for (i in ints) {
    if (i > 10) {
      trace(i)
    }
  }
  val chars = mutableListOf('a', 'b', 'c')
  chars.add('d') // 'add()' available
  chars += 'e'
  trace(chars)
  trace eq """
    13
    [a, b, c, d, e]
  """
}
```

Eine grundlegende `Liste` ist schreibgeschützt und enthält keine Änderungsfunktionen. Daher funktioniert die Änderungsfunktion `add()` nicht mit `ints`.

`for` Schleifen funktionieren gut mit `Listen`: `for(i in ints)` bedeutet, dass `i` jeden Wert in `ints` erhält.

`chars` wird als `VeränderbareListe` erstellt; sie kann mit Funktionen wie `add()` oder `remove()` modifiziert werden. Sie können auch `+=` und `-=` verwenden, um Elemente hinzuzufügen oder zu entfernen.

Eine schreibgeschützte `Liste` ist nicht dasselbe wie eine *unveränderliche* `Liste`, die überhaupt nicht modifiziert werden kann. Hier weisen wir `first`, eine veränderbare `Liste`, `second` zu, einer schreibgeschützten `Liste`-Referenz. Die schreibgeschützte Eigenschaft von `second` verhindert nicht, dass sich die `Liste` über `first` ändert:

```
// Summary2/MultipleListReferences.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first += 2
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first` und `second` verweisen auf dasselbe Objekt im Speicher. Wir verändern die `List` über die `first` Referenz und beobachten dann diese Änderung in der `second` Referenz.

Hier ist eine `List` von `Strings`, die durch das Aufteilen eines dreifach-quotierten Absatzes erstellt wurde. Dies zeigt die Leistungsfähigkeit einiger Funktionen der Standardbibliothek. Beachten Sie, wie diese Funktionen verkettet werden können:

```
// Summary2/ListOfStrings.kt
import atomictest.*

fun main() {
  val wocky = """
    Twas brillig, and the slithy toves
      Did gyre and gimble in the wabe:
    All mimsy were the borogoves,
      And the mome raths outgrabe.
  """.trim().split(Regex("\\W+"))
  trace(wocky.take(5))
  trace(wocky.slice(6..12))
  trace(wocky.slice(6..18 step 2))
  trace(wocky.sorted().takeLast(5))
  trace(wocky.sorted().distinct().takeLast(5))
  trace eq """
    [Twas, brillig, and, the, slithy]
    [Did, gyre, and, gimble, in, the, wabe]
    [Did, and, in, wabe, mimsy, the, And]
    [the, the, toves, wabe, were]
    [slithy, the, toves, wabe, were]
  """
}
```

`trim()` erzeugt einen neuen `String`, bei dem die führenden und nachfolgenden Leerzeichen (einschließlich Zeilenumbrüche) entfernt wurden. `split()` teilt den `String` gemäß seinem Argument. In diesem Fall verwenden wir ein `Regex`-Objekt, das einen *regulären Ausdruck* erstellt—ein Muster, das die zu trennenden Teile abgleicht. `\W` ist ein spezielles Muster, das "kein Wortzeichen" bedeutet, und `+` bedeutet "eines oder mehrere der vorhergehenden". Somit wird `split()` an einem oder mehreren Nicht-Wortzeichen brechen und somit den Textblock in seine einzelnen Wörter aufteilen.

In einem `String`-Literal steht `\` vor einem speziellen Zeichen und erzeugt zum Beispiel ein Zeilenumbruchzeichen (`\n`) oder ein Tabulatorzeichen (`\t`). Um einen tatsächlichen `\` im resultierenden String zu erzeugen, benötigen Sie zwei Backslashes: `"\\"`. Daher erfordern alle regulären Ausdrücke einen zusätzlichen `\`, um einen Backslash einzufügen, es sei denn, Sie verwenden einen dreifach zitierten `String`: `"""\W+"""`.

`take(n)` erzeugt eine neue `List`, die die ersten `n` Elemente enthält. `slice()` erzeugt eine neue `List`, die die durch das `Range`-Argument ausgewählten Elemente enthält,

und dieser `Range` kann einen `step` einschließen.

Beachten Sie den Namen `sorted()` anstelle von `sort()`. Wenn Sie `sorted()` aufrufen, *erzeugt* es eine sortierte `List` und lässt die ursprüngliche `List` unangetastet. `sort()` funktioniert nur mit einer `MutableList`, und diese Liste wird *vor Ort sortiert*—die ursprüngliche `List` wird verändert.

Wie der Name schon sagt, erzeugt `takeLast(n)` eine neue `List` der letzten `n` Elemente. An der Ausgabe können Sie sehen, dass "the" dupliziert ist. Dies wird durch Hinzufügen der `distinct()`-Funktion zur Aufrufkette beseitigt.

Parametrisierte Typen

Typparameter ermöglichen es uns, zusammengesetzte Typen zu beschreiben, am häufigsten Container. Insbesondere spezifizieren Typparameter, was ein Container enthält. Hier sagen wir Kotlin, dass `numbers` eine `List` von `Int` enthält, während `strings` eine `List` von `String` enthält:

```
// Summary2/ExplicitTyping.kt
package summary2
import atomictest.eq

fun main() {
  val numbers: List<Int> = listOf(1, 2, 3)
  val strings: List<String> =
    listOf("one", "two", "three")
  numbers eq "[1, 2, 3]"
  strings eq "[one, two, three]"
  toCharList("seven") eq "[s, e, v, e, n]"
}

fun toCharList(s: String): List<Char> =
  s.toList()
```

Sowohl bei den Definitionen von `numbers` als auch `strings` fügen wir Doppelpunkte und die Typdeklarationen `List<Int>` und `List<String>` hinzu. Die spitzen Klammern bezeichnen einen *Typ-Parameter*, der es uns ermöglicht zu sagen, "der Container enthält 'Parameter'-Objekte." Man spricht `List<Int>` typischerweise als "`List` von `Int`" aus.

Ein Rückgabewert kann ebenfalls einen Typ-Parameter haben, wie in `toCharList()` zu sehen ist. Man kann nicht einfach sagen, dass es eine `List` zurückgibt—Kotlin beschwert sich, also muss man den Typ-Parameter ebenfalls angeben.

Variable Argumentlisten

Das Schlüsselwort `vararg` steht für *variable Argumentliste* und erlaubt es einer Funktion, eine beliebige Anzahl von Argumenten (einschließlich null) des angegebenen Typs zu akzeptieren. Das `vararg` wird zu einem `Array`, das ähnlich wie eine `List` ist:

```
// Summary2/VarArgs.kt
package summary2
import atomictest.*

fun varargs(s: String, vararg ints: Int) {
  for (i in ints) {
    trace("$i")
  }
  trace(s)
}

fun main() {
  varargs("primes", 5, 7, 11, 13, 17, 19, 23)
  trace eq "5 7 11 13 17 19 23 primes"
}
```

Eine Funktionsdefinition kann nur einen Parameter als `vararg` spezifizieren. Jeder Parameter in der Liste kann das `vararg` sein, aber der letzte ist im Allgemeinen der einfachste.

Sie können ein `Array` von Elementen überall dort übergeben, wo ein `vararg` akzeptiert wird. Um ein `Array` zu erstellen, verwenden Sie `arrayOf()` auf die gleiche Weise wie `listOf()`. Ein `Array` ist immer veränderbar. Um ein `Array` in eine Sequenz von Argumenten (nicht nur ein einzelnes Element des Typs `Array`) zu konvertieren, verwenden Sie den *Spread-Operator* `*`:

```
// Summary2/ArraySpread.kt
import summary2.varargs
import atomictest.trace

fun main() {
  val array = intArrayOf(4, 5)      // [1]
  varargs("x", 1, 2, 3, *array, 6)  // [2]
  val list = listOf(9, 10, 11)
  varargs(
    "y", 7, 8, *list.toIntArray())  // [3]
  trace eq "1 2 3 4 5 6 x 7 8 9 10 11 y"
}
```

Wenn Sie ein `Array` von Primitivtypen wie im obigen Beispiel übergeben, muss die `Array`-Erstellungsfunktion spezifisch typisiert sein. Wenn **[1]** `arrayOf(4, 5)` anstelle von `intArrayOf(4, 5)` verwendet, erzeugt **[2]** einen Fehler: *inferred type is Array<Int> but IntArray was expected.*

Der Spread-Operator funktioniert nur mit Arrays. Wenn Sie eine `List` als Sequenz von Argumenten übergeben möchten, konvertieren Sie sie zuerst in ein `Array` und wenden Sie dann den Spread-Operator an, wie in **[3]**. Da das Ergebnis ein `Array` eines Primitivtyps ist, müssen wir die spezifische Konvertierungsfunktion `toIntArray()` verwenden.

Sets

`Sets` sind Sammlungen, die nur ein Element jedes Wertes zulassen. Ein `Set` verhindert automatisch Duplikate.

```
// Summary2/ColorSet.kt
package summary2
import atomictest.eq

val colors =
  "Yellow Green Green Blue"
    .split(Regex("""\W+""")).sorted()  // [1]

fun main() {
  colors eq
    listOf("Blue", "Green", "Green", "Yellow")
  val colorSet = colors.toSet()        // [2]
  colorSet eq
    setOf("Yellow", "Green", "Blue")
  (colorSet + colorSet) eq colorSet    // [3]
  val mSet = colorSet.toMutableSet()   // [4]
  mSet -= "Blue"
  mSet += "Red"                        // [5]
  mSet eq
    setOf("Yellow", "Green", "Red")
  // Set membership:
  ("Green" in colorSet) eq true        // [6]
  colorSet.contains("Red") eq false
}
```

- **[1]** Der `String` wird mit einem regulären Ausdruck aufgeteilt (`split()`), wie zuvor für `ListOfStrings.kt` beschrieben.
- **[2]** Wenn `colors` in das schreibgeschützte `Set` `colorSet` kopiert wird, wird einer der beiden `"Green"`-`Strings` entfernt, da es sich um ein Duplikat handelt.
- **[3]** Hier erstellen und anzeigen wir ein neues `Set` mit dem `+`-Operator. Das Einfügen von doppelten Elementen in ein `Set` entfernt diese Duplikate automatisch.
- **[4]** `toMutableSet()` erzeugt aus einem schreibgeschützten `Set` ein neues `MutableSet`.
- **[5]** Für ein `MutableSet` fügen die Operatoren `+=` und `-=` Elemente hinzu bzw. entfernen sie, wie sie es auch bei `MutableLists` tun.
- **[6]** Testen Sie die Mitgliedschaft in einem `Set` mit `in` oder `contains()`

Die normalen mathematischen Mengenoperationen wie Vereinigung, Schnittmenge, Differenz usw. sind alle verfügbar.

Maps

Ein `Map` verbindet *Schlüssel* mit *Werten* und sucht einen Wert anhand eines Schlüssels. Sie erstellen ein `Map`, indem Sie Schlüssel-Wert-Paare zu `mapOf()` bereitstellen. Mit `to` trennen wir jeden Schlüssel von seinem zugehörigen Wert:

```
// Summary2/ASCIIMap.kt
import atomictest.eq

fun main() {
  val ascii = mapOf(
    "A" to 65,
    "B" to 66,
    "C" to 67,
    "I" to 73,
    "J" to 74,
    "K" to 75
  )
  ascii eq
    "{A=65, B=66, C=67, I=73, J=74, K=75}"
  ascii["B"] eq 66                    // [1]
  ascii.keys eq "[A, B, C, I, J, K]"
  ascii.values eq
    "[65, 66, 67, 73, 74, 75]"
  var kv = ""
  for (entry in ascii) {              // [2]
    kv += "${entry.key}:${entry.value},"
  }
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  kv = ""
  for ((key, value) in ascii)         // [3]
    kv += "$key:$value,"
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  val mutable = ascii.toMutableMap() // [4]
  mutable.remove("I")
  mutable eq
    "{A=65, B=66, C=67, J=74, K=75}"
  mutable.put("Z", 90)
  mutable eq
    "{A=65, B=66, C=67, J=74, K=75, Z=90}"
  mutable.clear()
```

```
  mutable["A"] = 100
  mutable eq "{A=100}"
}
```

- **[1]** Ein Schlüssel (`"B"`) wird verwendet, um mit dem `[]` Operator einen Wert nachzuschlagen. Sie können alle Schlüssel mit `keys` und alle Werte mit `values` erzeugen. Der Zugriff auf `keys` erzeugt ein `Set`, da alle Schlüssel in einer `Map` bereits eindeutig sein müssen (ansonsten hätten Sie Mehrdeutigkeiten bei einem Nachschlagen).
- **[2]** Beim Iterieren durch eine `Map` entstehen Schlüssel-Wert-Paare als Map-Einträge.
- **[3]** Sie können Schlüssel-Wert-Paare während des Iterierens entpacken.
- **[4]** Sie können eine `MutableMap` aus einer Nur-Lese-`Map` mit `toMutableMap()` erstellen. Nun können wir Operationen durchführen, die `mutable` verändern, wie `remove()`, `put()`, und `clear()`. Eckige Klammern können ein neues Schlüssel-Wert-Paar in `mutable` zuweisen. Sie können auch ein Paar hinzufügen, indem Sie sagen `map += key to value`.

Eigenschafts-Accessoren

Der Zugriff auf die Eigenschaft `i` scheint unkompliziert:

```
// Summary2/PropertyReadWrite.kt
package summary2
import atomictest.eq

class Holder(var i: Int)

fun main() {
  val holder = Holder(10)
  holder.i eq 10 // Read the 'i' property
  holder.i = 20  // Write to the 'i' property
}
```

Allerdings ruft Kotlin Funktionen auf, um die Lese- und Schreiboperationen durchzuführen. Das Standardverhalten dieser Funktionen besteht darin, die in `i` gespeicherten Daten zu lesen und zu schreiben. Durch die Erstellung von *Eigenschaftszugriffen* ändern Sie die Aktionen, die beim Lesen und Schreiben auftreten.

Der Zugriff, der zum Abrufen des Werts einer Eigenschaft verwendet wird, wird als *Getter* bezeichnet. Um einen eigenen Getter zu erstellen, definieren Sie `get()` direkt nach der Eigenschaftsdeklaration. Der Zugriff, der zum Ändern einer veränderbaren Eigenschaft verwendet wird, wird als *Setter* bezeichnet. Um einen eigenen Setter zu erstellen, definieren Sie `set()` direkt nach der Eigenschaftsdeklaration. Die Reihenfolge der Definition von Gettern und Settern ist unwichtig, und Sie können einen ohne den anderen definieren.

Die Eigenschaftszugriffe im folgenden Beispiel imitieren die Standardimplementierungen und zeigen zusätzliche Informationen an, damit Sie sehen können, dass die Eigenschaftszugriffe tatsächlich während der Lese- und Schreibvorgänge aufgerufen werden. Wir rücken die `get()`- und `set()`-Funktionen ein, um sie visuell mit der Eigenschaft zu verknüpfen, aber die tatsächliche Verknüpfung erfolgt, weil sie direkt nach dieser Eigenschaft definiert sind:

```
// Summary2/GetterAndSetter.kt
package summary2
import atomictest.*

class GetterAndSetter {
  var i: Int = 0
    get() {
      trace("get()")
      return field
    }
    set(value) {
      trace("set($value)")
      field = value
    }
}

fun main() {
  val gs = GetterAndSetter()
  gs.i = 2
  trace(gs.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

Innerhalb des Getters und Setters wird der gespeicherte Wert indirekt mit dem `field`-Schlüsselwort manipuliert, das nur innerhalb dieser beiden Funktionen zugänglich ist. Es ist auch möglich, eine Eigenschaft zu erstellen, die kein `field` besitzt, sondern einfach den Getter aufruft, um ein Ergebnis zu erzeugen.

Wenn Sie eine `private` Eigenschaft deklarieren, werden beide Accessoren `private`. Sie können den Setter `private` und den Getter `public` machen. Das bedeutet, dass Sie die Eigenschaft außerhalb der Klasse lesen, aber ihren Wert nur innerhalb der Klasse ändern können.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Abschnitt III: Benutzerfreundlichkeit

Programmiersprachen unterscheiden sich nicht so sehr darin, was sie möglich machen, sondern darin, was sie einfach machen—**Larry Wall**, Erfinder der Perl-Sprache

Erweiterungsfunktionen

> Angenommen, Sie entdecken eine Bibliothek, die fast alles tut, was Sie brauchen… fast. Wenn sie nur ein oder zwei zusätzliche Mitgliedsfunktionen hätte, würde sie Ihr Problem perfekt lösen.

Aber es ist nicht Ihr Code – entweder haben Sie keinen Zugriff auf den Quellcode oder Sie kontrollieren ihn nicht. Sie müssten Ihre Änderungen bei jeder neuen Version wiederholen.

Kotlin's *extension functions* fügen bestehenden Klassen effektiv Mitgliedsfunktionen hinzu. Der Typ, den Sie erweitern, wird *receiver* genannt. Um eine Erweiterungsfunktion zu definieren, setzen Sie den Typ des Empfängers vor den Funktionsnamen:

```
fun ReceiverType.extensionFunction() { ... }
```

Dies fügt der `String`-Klasse zwei Erweiterungsfunktionen hinzu:

```
// ExtensionFunctions/Quoting.kt
package extensionfunctions
import atomictest.eq

fun String.singleQuote() = "'$this'"
fun String.doubleQuote() = "\"$this\""

fun main() {
  "Hi".singleQuote() eq "'Hi'"
  "Hi".doubleQuote() eq "\"Hi\""
}
```

Sie rufen Erweiterungsfunktionen auf, als ob sie Mitglieder der Klasse wären.

Um Erweiterungen aus einem anderen Paket zu verwenden, müssen Sie sie importieren:

```
// ExtensionFunctions/Quote.kt
package other
import atomictest.eq
import extensionfunctions.doubleQuote
import extensionfunctions.singleQuote

fun main() {
  "Single".singleQuote() eq "'Single'"
  "Double".doubleQuote() eq "\"Double\""
}
```

Sie können auf Mitgliederfunktionen oder andere Erweiterungen mit dem Schlüsselwort `this` zugreifen. `this` kann auch weggelassen werden, ebenso wie es innerhalb einer Klasse weggelassen werden kann, sodass Sie keine explizite Qualifizierung benötigen:

```
// ExtensionFunctions/StrangeQuote.kt
package extensionfunctions
import atomictest.eq

// Apply two sets of single quotes:
fun String.strangeQuote() =
  this.singleQuote().singleQuote()  // [1]

fun String.tooManyQuotes() =
  doubleQuote().doubleQuote()       // [2]

fun main() {
  "Hi".strangeQuote() eq "''Hi''"
  "Hi".tooManyQuotes() eq "\"\"Hi\"\""
}
```

- **[1]** `this` bezieht sich auf den `String`-Empfänger.
- **[2]** Wir lassen das Empfängerobjekt (`this`) beim ersten Aufruf der Funktion `doubleQuote()` weg.

Die Erweiterung Ihrer eigenen Klassen kann manchmal zu einfacherer Code führen:

```
// ExtensionFunctions/BookExtensions.kt
package extensionfunctions
import atomictest.eq

class Book(val title: String)

fun Book.categorize(category: String) =
  """title: "$title", category: $category"""

fun main() {
  Book("Dracula").categorize("Vampire") eq
    """title: "Dracula", category: Vampire"""
}
```

Innerhalb von `categorize()` greifen wir ohne explizite Qualifikation auf die `title`-Eigenschaft zu.

- -

Erweiterungsfunktionen können nur auf `öffentliche` Elemente des zu erweiternden Typs zugreifen. Daher können Erweiterungen dieselben Aktionen wie reguläre Funktionen ausführen. Sie können `Book.categorize(String)` als `categorize(Book, String)` umschreiben. Der einzige Grund für die Verwendung einer Erweiterungsfunktion ist die Syntax, aber dieser syntaktische Zucker ist mächtig. Für den aufrufenden Code sehen Erweiterungen genauso aus wie Mitgliedsfunktionen, und IDEs zeigen Erweiterungen an, wenn sie die Funktionen auflisten, die Sie für ein Objekt aufrufen können.

Übungen und Lösungen finden Sie unter www.AtomicKotlin.com.

Benannte & Standardargumente

Sie können während eines Funktionsaufrufs Argumentnamen angeben.

Benannte Argumente verbessern die Code-Lesbarkeit. Dies gilt besonders für lange und komplexe Argumentlisten – benannte Argumente können so klar sein, dass der Leser einen Funktionsaufruf verstehen kann, ohne die Dokumentation anzusehen.

In diesem Beispiel sind alle Parameter `Int`. Benannte Argumente verdeutlichen ihre Bedeutung:

```
// NamedAndDefaultArgs/NamedArguments.kt
package color1
import atomictest.eq

fun color(red: Int, green: Int, blue: Int) =
  "($red, $green, $blue)"

fun main() {
  color(1, 2, 3) eq "(1, 2, 3)"   // [1]
  color(
    red = 76,                     // [2]
    green = 89,
    blue = 0
  ) eq "(76, 89, 0)"
  color(52, 34, blue = 0) eq      // [3]
    "(52, 34, 0)"
}
```

- **[1]** Dies sagt Ihnen nicht viel. Sie müssen die Dokumentation einsehen, um zu verstehen, was die Argumente bedeuten.
- **[2]** Die Bedeutung jedes Arguments ist klar.
- **[3]** Es ist nicht erforderlich, alle Argumente zu benennen.

Benannte Argumente ermöglichen es Ihnen, die Reihenfolge der Farben zu ändern. Hier geben wir `blue` zuerst an:

```
// NamedAndDefaultArgs/ArgumentOrder.kt
import color1.color
import atomictest.eq

fun main() {
  color(blue = 0, red = 99, green = 52) eq
    "(99, 52, 0)"
  color(red = 255, 255, 0) eq
    "(255, 255, 0)"
}
```

Sie können benannte und reguläre (positionale) Argumente mischen. Wenn Sie die Reihenfolge der Argumente ändern, sollten Sie benannte Argumente im gesamten Aufruf verwenden—nicht nur der Lesbarkeit halber, sondern oft muss der Compiler wissen, wo die Argumente sind.

Benannte Argumente sind noch nützlicher, wenn sie mit *Standardargumenten* kombiniert werden, die Standardwerte für Argumente sind, die in der Funktionsdefinition angegeben sind:

```
// NamedAndDefaultArgs/Color2.kt
package color2
import atomictest.eq

fun color(
  red: Int = 0,
  green: Int = 0,
  blue: Int = 0,
) = "($red, $green, $blue)"

fun main() {
  color(139) eq "(139, 0, 0)"
  color(blue = 139) eq "(0, 0, 139)"
  color(255, 165) eq "(255, 165, 0)"
  color(red = 128, blue = 128) eq
    "(128, 0, 128)"
}
```

Jedes Argument, das Sie nicht angeben, erhält seinen Standardwert. Daher müssen Sie nur die Argumente angeben, die von den Standardwerten abweichen. Wenn Sie eine lange Argumentliste haben, vereinfacht dies den resultierenden Code, was das Schreiben und—was noch wichtiger ist—das Lesen erleichtert.

Dieses Beispiel verwendet auch ein *nachgestelltes Komma* in der Definition von `color()`. Das nachgestellte Komma ist das zusätzliche Komma nach dem letzten Parameter (`blue`). Dies ist nützlich, wenn Ihre Parameter oder Werte über mehrere Zeilen geschrieben sind. Mit einem nachgestellten Komma können Sie neue Elemente hinzufügen und ihre Reihenfolge ändern, ohne Kommas hinzuzufügen oder zu entfernen.

Benannte und Standardargumente (sowie nachgestellte Kommas) funktionieren auch für Konstruktoren:

```
// NamedAndDefaultArgs/Color3.kt
package color3
import atomictest.eq

class Color(
  val red: Int = 0,
  val green: Int = 0,
  val blue: Int = 0,
) {
  override fun toString() =
    "($red, $green, $blue)"
}

fun main() {
  Color(red = 77).toString() eq "(77, 0, 0)"
}
```

`joinToString()` ist eine Standardbibliotheksfunktion, die Standardargumente verwendet. Sie kombiniert die Inhalte eines iterierbaren Objekts (einer Liste, Menge oder eines Bereichs) zu einem `String`. Sie können einen Trennzeichen, ein Präfixelement und ein Suffixelement angeben:

```
// NamedAndDefaultArgs/CreateString.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3,)
  list.toString() eq "[1, 2, 3]"
  list.joinToString() eq "1, 2, 3"
  list.joinToString(prefix = "(",
    postfix = ")") eq "(1, 2, 3)"
  list.joinToString(separator = ":") eq
    "1:2:3"
}
```

Der Standardwert von `toString()` für eine `List` gibt den Inhalt in eckigen Klammern zurück, was möglicherweise nicht das ist, was Sie wollen. Die Standardwerte für die Parameter von `joinToString()` sind ein Komma für `separator` und leere `Strings` für `prefix` und `postfix`. Im obigen Beispiel verwenden wir benannte und Standardargumente, um nur die Argumente zu spezifizieren, die wir ändern möchten.

Der Initialisierer für `list` beinhaltet ein abschließendes Komma. Normalerweise verwenden Sie ein abschließendes Komma nur, wenn jedes Element in einer eigenen Zeile steht.

Wenn Sie ein Objekt als Standardargument verwenden, wird bei jedem Aufruf eine neue Instanz dieses Objekts erstellt:

Wenn Sie eine Objektinstanz als Standardargument übergeben (`da` innerhalb von `g()` im folgenden Beispiel), wird dieselbe Instanz für jeden Aufruf von `g()` verwendet. Wenn Sie die Syntax für einen Konstruktoraufruf übergeben (`DefaultArg()` innerhalb von `h()`), wird dieser Konstruktor jedes Mal aufgerufen, wenn Sie `h()` aufrufen:

```
// NamedAndDefaultArgs/Evaluation.kt
package namedanddefault

class DefaultArg
val da = DefaultArg()

fun g(d: DefaultArg = da) = println(d)

fun h(d: DefaultArg = DefaultArg()) =
  println(d)

fun main() {
  g()
  g()
  h()
  h()
}
/* Sample output:
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@49476842
namedanddefault.DefaultArg@78308db1
*/
```

Die Ausgabe der beiden `g()` Aufrufe zeigt identische Objektadressen. Bei den beiden Aufrufen von `h()` sind die Adressen der `DefaultArg` Objekte unterschiedlich, was zeigt, dass es zwei verschiedene Objekte gibt.

Geben Sie Argumentnamen an, wenn sie die Lesbarkeit verbessern. Vergleichen Sie die folgenden beiden Aufrufe von `joinToString()`:

```
// NamedAndDefaultArgs/CreateString2.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3)
  list.joinToString(". ", "", "!") eq
    "1. 2. 3!"
  list.joinToString(separator = ". ",
    postfix = "!") eq "1. 2. 3!"
}
```

Es ist schwer zu erraten, ob ". " oder "" ein Trennzeichen ist, es sei denn, man merkt sich die Reihenfolge der Parameter, was unpraktisch ist.

Ein weiteres Beispiel für Standardargumente ist die `trimMargin()`-Funktion der Standardbibliothek, die mehrzeilige `String`s formatiert. Sie verwendet einen Randpräfix-`String`, um den Anfang jeder Zeile festzulegen. `trimMargin()` entfernt führende Leerzeichen, gefolgt von dem Randpräfix, aus jeder Zeile des Quell-`String`. Es entfernt die erste und letzte Zeile, wenn sie leer sind:

```
// NamedAndDefaultArgs/TrimMargin.kt
import atomictest.eq

fun main() {
  val poem = """
    |->Last night I saw upon the stair
        |->A little man who wasn't there
          |->He wasn't there again today
|->Oh, how I wish he'd go away."""
  poem.trimMargin() eq
"""->Last night I saw upon the stair
->A little man who wasn't there
->He wasn't there again today
->Oh, how I wish he'd go away."""
  poem.trimMargin(marginPrefix = "|->") eq
"""Last night I saw upon the stair
A little man who wasn't there
He wasn't there again today
Oh, how I wish he'd go away."""
}
```

Das | ("Pipe") ist das Standardargument für das Randpräfix, und Sie können es durch einen `String` Ihrer Wahl ersetzen.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.

Überladung

> Sprachen ohne Unterstützung für Standardargumente verwenden oft Überladung, um dieses Merkmal zu imitieren.

Der Begriff *Überladung* bezieht sich auf den Namen einer Funktion: Sie verwenden denselben Namen ("überladen" diesen Namen) für verschiedene Funktionen, solange sich die Parameterlisten unterscheiden. Hier überladen wir die Memberfunktion `f()`:

```
// Overloading/Overloading.kt
package overloading
import atomictest.eq

class Overloading {
  fun f() = 0
  fun f(n: Int) = n + 2
}

fun main() {
  val o = Overloading()
  o.f() eq 0
  o.f(11) eq 13
}
```

In `Overloading` sehen Sie zwei Funktionen mit demselben Namen, `f()`. Die *Signatur* einer Funktion besteht aus dem Namen, der Parameterliste und dem Rückgabetyp. Kotlin unterscheidet eine Funktion von einer anderen, indem es die Signaturen vergleicht. Beim Überladen von Funktionen müssen die Parameterlisten einzigartig sein—man kann nicht nur über die Rückgabetypen überladen.

Die Aufrufe zeigen, dass es sich tatsächlich um unterschiedliche Funktionen handelt. Eine Funktionssignatur beinhaltet auch Informationen über die umschließende Klasse (oder den Empfangstyp, wenn es sich um eine Erweiterungsfunktion handelt).

Wenn eine Klasse bereits eine Mitgliedsfunktion mit derselben Signatur wie eine Erweiterungsfunktion hat, bevorzugt Kotlin die Mitgliedsfunktion. Sie können jedoch die Mitgliedsfunktion mit einer Erweiterungsfunktion überladen:

```
// Overloading/MemberVsExtension.kt
package overloading
import atomictest.eq

class My {
  fun foo() = 0
}

fun My.foo() = 1              // [1]

fun My.foo(i: Int) = i + 2    // [2]

fun main() {
  My().foo() eq 0
  My().foo(1) eq 3
}
```

- **[1]** Es ist sinnlos, eine Erweiterung zu deklarieren, die ein Mitglied dupliziert, da sie niemals aufgerufen werden kann.
- **[2]** Sie können eine Mitgliedsfunktion mit einer Erweiterungsfunktion überladen, indem Sie eine andere Parameterliste bereitstellen.

Verwenden Sie das Überladen nicht, um Standardargumente zu imitieren. Das heißt, tun Sie dies nicht:

```
// Overloading/WithoutDefaultArguments.kt
package withoutdefaultarguments
import atomictest.eq

fun f(n: Int) = n + 373
fun f() = f(0)

fun main() {
  f() eq 373
}
```

Die Funktion ohne Parameter ruft einfach die erste Funktion auf. Die beiden Funktionen können durch eine einzelne Funktion ersetzt werden, indem ein Standardargument verwendet wird:

```
// Overloading/WithDefaultArguments.kt
package withdefaultarguments
import atomictest.eq

fun f(n: Int = 0) = n + 373

fun main() {
  f() eq 373
}
```

In beiden Beispielen können Sie die Funktion entweder ohne ein Argument oder durch Übergeben eines Ganzzahlwerts aufrufen. Bevorzugen Sie die Form in `WithDefaultArguments.kt`.

Bei der Verwendung von überladenen Funktionen zusammen mit Standardargumenten sucht der Aufruf der überladenen Funktion nach der "nächsten" Übereinstimmung. Im folgenden Beispiel ruft der `foo()`-Aufruf in `main()` *nicht* die erste Version der Funktion mit ihrem Standardargument von 99 auf, sondern stattdessen die zweite Version, die ohne Parameter:

```
// Overloading/OverloadedVsDefaultArg.kt
package overloadingvsdefaultargs
import atomictest.*

fun foo(n: Int = 99) = trace("foo-1-$n")

fun foo() {
  trace("foo-2")
  foo(14)
}

fun main() {
  foo()
  trace eq """
    foo-2
    foo-1-14
  """
}
```

Sie können das Standardargument 99 niemals nutzen, da `foo()` immer die zweite Version von `f()` aufruft.

Warum ist das Überladen nützlich? Es ermöglicht Ihnen, "Variationen eines Themas" klarer auszudrücken, als wenn Sie gezwungen wären, unterschiedliche Funktionsnamen zu verwenden. Angenommen, Sie möchten Additionsfunktionen:

```
// Overloading/OverloadingAdd.kt
package overloading
import atomictest.eq

fun addInt(i: Int, j: Int) = i + j
fun addDouble(i: Double, j: Double) = i + j

fun add(i: Int, j: Int) = i + j
fun add(i: Double, j: Double) = i + j

fun main() {
  addInt(5, 6) eq add(5, 6)
  addDouble(56.23, 44.77) eq
    add(56.23, 44.77)
}
```

`addInt()` nimmt zwei `Int`s und gibt ein `Int` zurück, während `addDouble()` zwei `Double`s nimmt und ein `Double` zurückgibt. Ohne Überladen kann man die Operation nicht einfach `add()` nennen, daher kombinieren Programmierer typischerweise *was* mit *wie*, um eindeutige Namen zu erzeugen (man kann auch eindeutige Namen mit zufälligen Zeichen erstellen, aber das typische Muster ist die Verwendung von aussagekräftigen Informationen wie Parametertypen). Im Gegensatz dazu ist das überladene `add()` viel klarer.

- -

Das Fehlen des Überladens in einer Sprache ist keine große Bürde, aber das Feature bietet wertvolle Vereinfachung, wodurch der Code lesbarer wird. Mit Überladung sagt man einfach *was*, was die Abstraktionsebene erhöht und die geistige Belastung für den Leser verringert. Wenn man wissen will *wie*, schaut man sich die Parameter an. Beachten Sie auch, dass Überladen Redundanz reduziert: Wenn wir `addInt()` und `addDouble()` sagen müssen, wiederholen wir im Wesentlichen die Parameterinformationen im Funktionsnamen.

Übungen und Lösungen finden Sie auf www.AtomicKotlin.com.