DAS SWIFT HANDBUCH

Thomas SILLMANN

3. Auflage

Apps programmieren für **macOS, iOS, watchOS** und **tvOS**

Praxisprojekte unter:
plus.hanser-fachbuch.de

ZU VERSION: Swift 5.9

HANSER

Sillmann

Das Swift-Handbuch

Ihr Plus – digitale Zusatzinhalte!

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

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

plus-M37rq-aN56m

Bleiben Sie auf dem Laufenden!

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

www.hanser-fachbuch.de/newsletter

Thomas Sillmann

Das Swift-Handbuch

Apps programmieren für macOS, iOS, watchOS und tvOS

3., aktualisierte Auflage

HANSER

Der Autor: *Thomas Sillmann*, Aschaffenburg
www.thomassillmann.de

Bibliografische Information der Deutschen Nationalbibliothek:

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

Lektorat: Sylvia Hasselbach
Copy editing: Walter Saumweber, Ratingen
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Titelmotiv: Tom West, unter Verwendung von Grafiken von © Max Kostopoulos
Satz: Eberl & Koesel Studio, Kempten
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN: 978-3-446-47639-4
E-Book-ISBN: 978-3-446-47857-2
E-Pub-ISBN: 978-3-446-48032-2

„Bis zum Mond und wieder zurück haben wir uns lieb.“

Für meinen Vater

Wo Du auch bist, Du begleitest mich auf meiner Reise –
bis zu jenem Tag, an dem wir uns wiedersehen
und uns viele Geschichten erzählen werden.

Inhalt

Vorwort

Herzlich Willkommen in der Welt von Swift, Apples hauseigener Programmiersprache zur Entwicklung von Apps für iPhone, iPad, Mac und Co.! Da Sie dieses Buch in Händen halten, mutmaße ich, dass Sie mehr über die Programmierung für Apple-Plattformen erfahren oder Ihre ersten Schritte in dieser faszinierenden Welt bestreiten möchten.

Bevor ich Sie in die Welt der Programmierung entführe, möchte ich dieses Vorwort dazu nutzen, einige grundlegende Worte über den Aufbau und die Inhalte dieses Buches zu verlieren. Das soll Ihnen als erste Übersicht und weiterer Wegweiser dienen, um bestmöglich mit dem Buch arbeiten zu können und schnelle Erfolge bei der Programmierung zu erzielen.

Inhalte

Das Buch basiert auf der im Herbst 2023 erschienenen Version 5.9 von Swift sowie der Version 15 der Entwicklungsumgebung Xcode. Alle Kapitel der vorherigen Auflage wurden entsprechend aktualisiert sowie um neue Inhalte ergänzt. Zu den Highlights dieser Neuerungen zählt unter anderem das SwiftData-Framework für die persistente Speicherung von Daten, die neuen String Catalogs zur Übersetzung von Apps sowie das `Observable`-Makro als Alternative zum `ObservedObject`-Property Wrapper.

Generell schlägt diese dritte Auflage des Swift-Handbuchs inhaltlich den gleichen Weg ein wie die vorherige Auflage. Der Fokus liegt klar auf den drei wichtigsten Elementen für die App-Entwicklung für macOS, iOS/iPadOS, watchOS und tvOS:

- Die Programmiersprache Swift
- Die Entwicklungsumgebung Xcode
- Die App-Entwicklung mit SwiftUI

Insbesondere SwiftUI ist in dieser Auflistung hervorzuheben. Seit der erstmaligen Vorstellung im Jahr 2019 hat sich SwiftUI massiv weiterentwickelt und bietet inzwischen eine Vielzahl an Möglichkeiten zur Umsetzung und Gestaltung von Apps.

Dieses Handbuch liefert Ihnen handfestes Wissen zu allen drei genannten Bereichen. So lernen Sie die Programmiersprache Swift und ihre verschiedenen Sprachfeatures kennen. Sie erfahren, wie die Entwicklungsumgebung Xcode aufgebaut ist und welche Funktionen Ihnen bei der täglichen Entwicklerarbeit zur Verfügung stehen. Und Sie erhalten einen Überblick über die Möglichkeiten, die Sie zur Gestaltung von Apps mit SwiftUI nutzen können. Darüber hinaus finden Sie auch Kapitel zur Versionsverwaltung und zum Testing sowie zur Veröffentlichung von Apps im App Store.

Kurzum: Dieses Handbuch soll Ihnen als Grundlage dienen, um einen Überblick über die wichtigsten Bereiche der App-Entwicklung für Apple-Plattformen zu erhalten und Ihnen das nötige Verständnis vermitteln, um das erlangte Wissen in eigenen Projekten zum Einsatz bringen zu können.

Aufbau des Buches

Mir ist es wichtig, dass Sie das Buch als Referenzwerk nutzen können. Entsprechend finden Sie je einen eigenen Teil zur Programmiersprache Swift, zur Entwicklungsumgebung Xcode sowie zur App-Entwicklung mit SwiftUI. Das ermöglicht es Ihnen, sich separat mit diesen drei Bestandteilen auseinanderzusetzen und notwendige Infos zu erhalten, ohne diese Inhalte im Buch zu vermischen.

Ich möchte Sie in die Lage versetzen, eigene Apps entwickeln zu können. Zu diesem Zweck halte ich die Code-Beispiele im Buch bewusst klein und setze nicht auf die Umsetzung ganzer Projekte. Stattdessen sollen Sie das nötige Wissen erlangen, um Ihre eigenen Anwendungen erstellen zu können und zu diesem Zweck alles über die dafür notwendigen Komponenten erfahren.

Ein solches Grundverständnis erlaubt es Ihnen außerdem, sich selbstständig in weitere Bereiche einzuarbeiten und künftige Neuerungen mithilfe der Dokumentation anzuwenden.

Beispielprojekte auf Hanser-Plus

Über Hanser-Plus stehen Ihnen kleine Beispielprojekte zum Download zur Verfügung, die einzelne Aspekte der App-Entwicklung und der Programmierung mit Swift beleuchten. Diese Beispiele sind bewusst überschaubar gehalten, um den Fokus auf grundlegende Funktionsweisen zu richten und Ihnen einen verständlichen Überblick zu verschaffen. Nutzen Sie diese Beispiele gerne, um darauf aufbauend selbst ein wenig im Code zu experimentieren oder um zu überprüfen, wie sich bestimmte Funktionen umsetzen lassen. Um die Beispiele herunterzuladen, geben Sie auf der Webseite

https://plus.hanser-fachbuch.de/

den Code

```
plus-M37rq-aN56m
```

ein.

Feedback

In diesem Sinne wünsche ich Ihnen nun viel Freude mit dem Buch und Erfolg beim Programmieren. Sollten Sie Feedback zum Buch haben, können Sie mich gerne via E-Mail an

contact@thomassillmann.de

kontaktieren.

Ich bin inzwischen seit über zehn Jahren begeisterter Entwickler für die verschiedenen Apple-Plattformen. Ich finde das Apple-Ökosystem, die Programmiersprache Swift und die Entwicklungsumgebung Xcode faszinierend und arbeite jeden Tag voller Begeisterung damit. Ich hoffe, dass ich mit diesem Buch einen Teil meiner Begeisterung auch auf Sie übertragen kann.

Thomas Sillmann

September 2023

Teil I: Swift

1 Die Programmiersprache Swift

Nach dem Vorwort heiße ich Sie nun an dieser Stelle ein weiteres Mal recht herzlich willkommen in der faszinierenden Welt von Swift. 😊 Bevor es im zweiten Kapitel konkret mit der Programmierung losgeht, möchte ich Ihnen an dieser Stelle etwas über die Geschichte von Swift, die Bedeutung der Sprache im Apple-Kosmos sowie die Tools erzählen, die für uns als Entwickler unabdingbar sind.

1.1 Die Geschichte von Swift

Keine Bange, das soll hier keine langatmige monotone Geschichtsstunde werden. Falls es Sie auch nicht im Geringsten interessieren sollte, wie Swift entstanden ist, dürfen Sie diesen Abschnitt guten Gewissens gerne überspringen; ich wäre Ihnen nicht böse deswegen. 😊 Obwohl ich den Hintergrund, wie Swift das Licht der Welt erblickte, durchaus spannend finde.

Dabei sind viele Details über die genaue Entstehungsgeschichte von Swift gar nicht bekannt. Was man weiß, ist, dass Chris Lattner wohl in gewisser Weise als „Vater“ von Swift bezeichnet werden kann. Er begann die Entwicklung von Swift im Juli 2010 bei Apple aus eigenem Antrieb heraus und zunächst im Alleingang. Ab Ende 2011 kamen dann weitere Entwickler dazu, währwend das Projekt im Geheimen bei Apple fortgeführt wurde. Das erste Mal zeigte Apple die neue Sprache der Weltöffentlichkeit auf der WWDC (Worldwide Developers Conference) 2014 (siehe Bild 1.1).

Bild 1.1 Auf der WWDC 2014 präsentierte Apple Swift erstmals der Weltöffentlichkeit.

Mit dieser erstmaligen Präsentation von Swift überraschte Apple sowohl Presse als auch Entwickler gleichermaßen. Dabei war die Sprache zunächst – ähnlich wie Objective-C – ausschließlich auf die Plattformen von Apple beschränkt. Ein Mac mitsamt der zugehörigen IDE Xcode von Apple waren also Pflicht, wollte man mit Swift Apps für macOS, iOS, watchOS oder tvOS entwickeln. Im Herbst 2014 folgte dann die erste finale Version von Swift, die Apple den Entwicklern zusammen mit einem Update für Xcode zugänglich machte.

Im darauffolgenden Jahr sorgte Apple auf der WWDC 2015 dann für die nächste große Überraschung. Sie präsentierten nicht nur die neue Version 2 von Swift, sondern gaben auch bekannt, dass Swift noch im gleichen Jahr Open Source werden würde. Dieses Versprechen wurde dann am 03. Dezember 2015 umgesetzt und Apple startete die Plattform , um darüber zukünftig alle Weiterentwicklungen und Neuerungen zu Swift zusammenzutragen.

Seitdem hat die Sprache viele weitere Versionssprünge hinter sich und sie entwickelt sich noch immer stetig weiter. Einen großen Beitrag leistet hierbei auch die große Swift-Community, die mögliche Neuerungen und Änderungen vorantreibt.

Heute ist Swift die Sprache der Wahl, wenn es um die Entwicklung von Apps für Apple-Plattformen geht. Darüber hinaus findet Swift aber auch auf immer mehr Systemen wie Linux oder Windows Verwendung, ist dort aber bei weitem nicht so verbreitet wie im Apple-Umfeld.

1.2 Die Bedeutung von Swift im Apple-Kosmos

Aus Sicht von Apple ist eines ganz offensichtlich: Swift gehört die Zukunft. Zwar unterstützt Apple noch immer die „alte“ Programmiersprache Objective-C, doch die wird im Gegensatz zu Swift kaum bis gar nicht weiterentwickelt.

Auch ist Swift in bestimmten Bereichen der App-Entwicklung inzwischen ein Muss. Manche Systemfunktionen lassen sich nur mit Swift und nicht mit Objective-C ansteuern. So zeigt Apple auf sehr drastische Art und Weise, welcher Programmiersprache Entwickler ihre Aufmerksamkeit widmen sollten.

Zwar gibt es noch sehr viel Objective-C-Code da draußen und sicherlich diverse Entwickler-Kollegen, die Objective-C im Vergleich zu Swift klar bevorzugen. Doch es ändert nichts daran, dass Swift die Zukunft gehört, und danach sollten auch wir Entwickler uns richten.

Und wenn ich hier persönlich werden darf: In meinen Augen ist Swift nicht nur eine äußerst mächtige und vielseitige, sondern auch wunderschön zu schreibende Programmiersprache. Mit modernen Ansätzen erleichtert sie außerdem Neulingen den Einstieg, und ich behaupte einmal, dass diese Aussage jeder unterschreiben kann, der bereits einmal mit Objective-C entwickelt hat.

Auch Swift mag nicht perfekt sein, doch die Entwicklung der letzten Jahre zeigt, dass die Sprache auf einem verdammt guten Weg ist und sowohl von Apple als auch der Open-Source-Community sehr viel Pflege erfährt.

Es mag abgedroschen klingen, doch *jetzt* ist definitiv der beste Zeitpunkt, sich der Programmierung mit Swift zu widmen.

1.3 Das UI-Framework: SwiftUI

In Teil 3 dieses Buches, der sich vollumfassend den spezifischen Besonderheiten der App-Entwicklung widmet, werden Sie sehr sehr viel über SwiftUI lesen. Bei SwiftUI handelt es sich um ein sogenanntes *Framework*. Frameworks setzen sich aus einem Set verschiedener Funktionen zusammen, die Sie als App-Entwickler nutzen können, wenn Sie das Framework in Ihr App-Projekt einbinden.

Auch wenn SwiftUI nicht direkt etwas mit der Programmiersprache Swift zu tun hat, möchte ich an dieser Stelle ein paar Worte darüber verlieren. Das hängt nämlich mit einer grundsätzlichen Verständnisfrage zusammen, auf die ich in den letzten Jahren öfters gestoßen bin. So gilt:

Swift ist eine Programmiersprache. SwiftUI ist ein Framework, das auf Swift basiert und spezielle Funktionen für die Erstellung von Nutzeroberflächen zur Verfügung stellt.

Wenn Sie also die Begriffe Swift und SwiftUI hören, geht es nicht darum, welches der beiden Sie verwenden sollen. Wenn Sie SwiftUI einsetzen, kommen Sie um ein Verständnis der Programmiersprache Swift gar nicht herum, denn darauf basiert SwiftUI nun einmal. Doch beschränkt sich der Einsatz von Swift nicht nur auf die Entwicklung von Nutzeroberflächen. So gesehen benötigen Sie Swift *immer*. Es ist die grundlegendste Säule der App-Entwicklung.

Wie gesagt, finden Sie in Teil 3 des Buches weitreichende Informationen zur Nutzung und Funktionsweise von SwiftUI.

1.4 Was Sie als App-Entwickler brauchen

Um mit Swift Ihre eigenen Apps für iPhone, iPad und Co. entwickeln zu können, führt in der Regel kein Weg an Apples hauseigener Entwicklungsumgebung Xcode vorbei (warum ich hier „in der Regel“ schreibe, erläutere ich noch).

Xcode ist eine App, die exklusiv auf dem Mac zur Verfügung steht und die Sie kostenlos und bequem aus dem Mac App Store herunterladen und installieren können (siehe Bild 1.2).

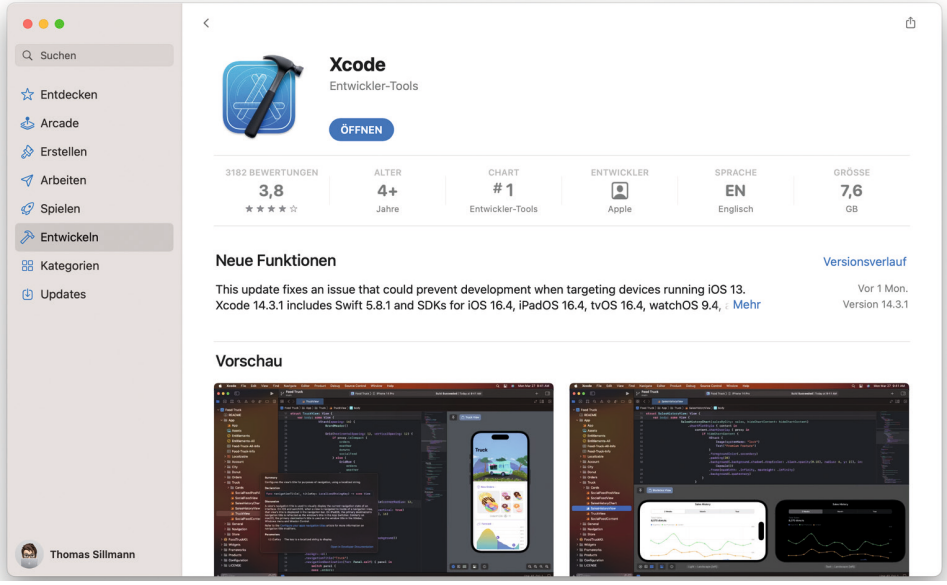

Bild 1.2 Xcode ist Apples vollwertige Entwicklungsumgebung für die Programmierung von Apps.

Xcode bringt alles mit, was Sie für die App-Entwicklung benötigen. Dazu gehört auch Unterstützung für Swift und SwiftUI. Außerdem verfügt Xcode über verschiedene Simulatoren, die es Ihnen beispielsweise ermöglichen, iPhone-Apps direkt auf Ihrem Mac auszuführen und zu testen.

Mithilfe von Xcode erstellen Sie Projekte für Ihre Apps und verwalten darin sowohl den Quellcode als auch weitere Ressourcen wie Bilder. Xcode kann Ihnen darüber hinaus beim Auffinden von Fehlern in Ihrer App helfen und ermöglicht es sogar, Ihre fertige App direkt in den App Store hochzuladen. Mehr zur Veröffentlichung von Apps erfahren Sie im letzten Teil dieses Buches.

Aufgrund dieses großen Funktionsumfangs kann Xcode als durchaus komplexe Anwendung angesehen werden. Das gilt umso mehr, wenn man zuvor noch keinerlei Programmiererfahrung gesammelt hat. Aus diesem Grund liefert Ihnen der zweite Teil dieses Buches einen Rundumüberblick über den Aufbau und die verschiedenen Bestandteile von Apples Entwicklungsumgebung.

Die abgespeckte Xcode-Alternative: Swift Playgrounds

Ich habe ja zu Beginn dieses Abschnitts geschrieben, dass zur App-Entwicklung „in der Regel“ kein Weg an Xcode vorbeiführt. Tatsächlich gibt es für das iPad eine Alternative zur vollwertigen Entwicklungsumgebung auf dem Mac: die App *Swift Playgrounds* (siehe Bild 1.3).

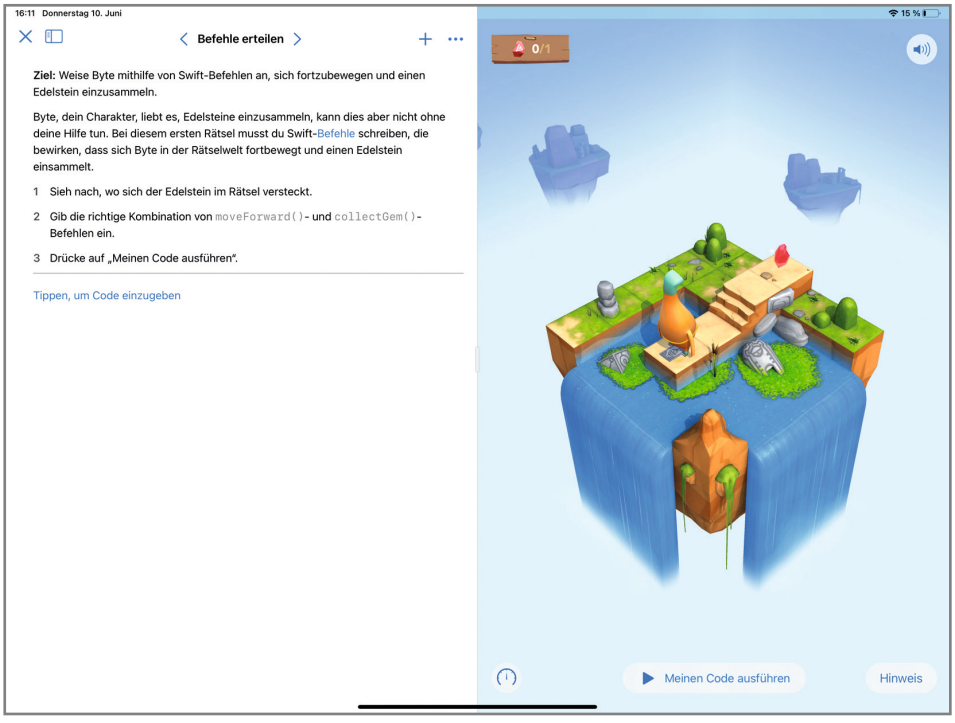

Bild 1.3 Swift Playgrounds ermöglicht das Programmieren auch auf Apples iPad.

Swift Playgrounds entstand ursprünglich, um Nutzern die Grundlagen der Programmierung spielerisch näherzubringen. Diese Funktion besitzt die App auch heute noch und sie ist definitiv einen Blick wert. Zusätzlich ermöglicht sie es inzwischen aber auch, vollwertige Apps für iPhone und iPad mit ihr zu entwickeln und direkt in den App Store hochzuladen.

Das ist definitiv eine großartige Sache, doch sollte man sich keinen Illusionen hingeben: Für die professionelle Entwicklung von Apps ist noch immer Xcode das Mittel der Wahl. Xcode bietet deutlich mehr Funktionen und Möglichkeiten als Swift Playgrounds. Dafür erlaubt es Swift Playgrounds, Apps nicht nur auf dem Mac, sondern auch auf Apples iPad zu entwickeln.

Übrigens: Für die App-Entwicklung unterstützt Swift Playgrounds ausschließlich die Programmiersprache Swift sowie das neue UI-Framework SwiftUI. Das ist also ein klarer Grund mehr, sich heute ausgiebig mit diesen beiden so wichtigen Technologien für Apple-Entwickler auseinanderzusetzen.

1.5 Programmieren für Beginner (und darüber hinaus): Playgrounds

Wenn es um die Entwicklung von Apps geht, arbeitet man in sogenannten *Projekten*. In diesen verwaltet man neben dem Quellcode auch alle zusätzlichen Ressourcen wie Bilder. Außerdem enthält solch ein Projekt weitere Informationen wie die Versionsnummer und die Unterstützung für optionale Services (beispielsweise iCloud).

Gerade am Anfang erhält man mit solch einem Projekt von Haus aus bereits sehr viele Dateien, die einen gerade als Anfänger überfordern können. Außerdem möchte man in manchen Fällen auch einfach einmal ein wenig mit Swift experimentieren, ohne dafür gleich ein ganzes Projekt erzeugen zu müssen.

Abhilfe schaffen hier die sogenannten *Playgrounds* (siehe Bild 1.4). In einem Playground können Sie einfach Code drauflos schreiben, ohne sich mit Projektstrukturen und dem Ablauf Ihrer App beschäftigen zu müssen. Per Klick auf einen Button führen Sie den Code aus und sehen umgehend dessen Ergebnis.

Gerade zu Beginn ist es absolut sinnvoll, Ihren Code in Playgrounds zu schreiben und zu testen. So machen Sie sich schnell mit den Grundlagen und der Syntax von Swift vertraut, ohne von den genannten Projektstrukturen ganzer Apps womöglich überfordert zu werden.

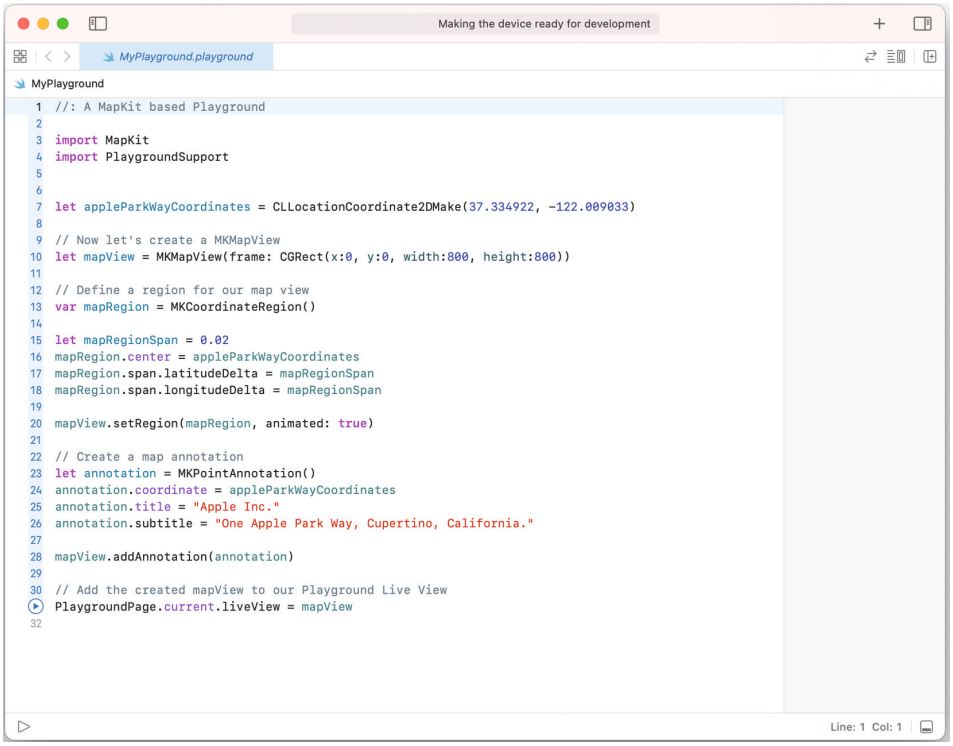

Bild 1.4 Playgrounds sind ein ideales Mittel, um zu experimentieren und das Programmieren zu lernen.

Gerade wenn Sie bisher nur wenig oder gar keine Erfahrung mit Swift gesammelt haben, empfehle ich Ihnen, die Beispiele in diesem ersten Teil des Buches mithilfe von Playgrounds ebenfalls zu programmieren. Nutzen Sie die Flexibilität von Playgrounds, um ruhig auch selbst zu experimentieren und Dinge auszuprobieren, die Sie interessieren oder die Ihnen in den Sinn kommen. Besser kann man sich mit der Programmierung gar nicht vertraut machen!

Übrigens sind Playgrounds nicht nur ein ideales Mittel, um das Programmieren grundlegend zu lernen. Auch Profis nutzen sie, um sich bequem mit neuen Frameworks oder der Umsetzung einer komplexen Programmlogik auseinanderzusetzen. Entwickler können so Ihren Code unabhängig von einem vollständigen und meist komplexen App-Projekt testen. Hat man eine passende Lösung erarbeitet, kann man diese anschließend in das eigentliche Projekt übertragen.

Mehr zur Verwendung von Playgrounds und wie Sie solche mit Xcode erstellen, erfahren Sie im zweiten Teil dieses Buches.

Swift Playgrounds auf dem iPad

Sie können Playgrounds auch auf dem iPad erstellen und so auf Apples Tablet programmieren. Zu diesem Zweck steht Ihnen eine App mit dem passenden Namen *Swift Playgrounds* zur Verfügung, die Sie kostenlos aus dem App Store laden können (siehe auch Abschnitt 1.4, „Was Sie als App-Entwickler brauchen“).

Die App eignet sich auch ideal für Einsteiger, die das Programmieren lernen möchten. Sie finden darin verschiedene kleine Lernkurse, die spielerisch die Grundlagen der Entwicklung mit Swift vorstellen. Darüber hinaus können Sie mit der App aber auch die eben beschriebenen Playgrounds erstellen und so ganz frei mit Code und dem Programmieren experimentieren.

Übrigens steht die Swift-Playgrounds-App auch als dedizierte Anwendung auf dem Mac zur Verfügung, unabhängig von Xcode. Falls Sie die genannten Lernkurse daher auch auf dem Mac ausprobieren möchten, können Sie das über die Swift-Playgrounds-App tun. Wenn Sie aber erst einmal näher mit der Swift-Programmierung und der Entwicklungsumgebung Xcode vertraut sind, benötigen Sie die App nicht wirklich auf dem Mac. Xcode liefert Ihnen alles, was Sie brauchen, um Ihre eigenen Anwendungen umzusetzen und Code in separaten Playgrounds zu testen.

1.6 Weitere wichtige Ressourcen

In der Welt der App-Entwicklung gibt es keinen Stillstand. In mehr oder weniger regelmäßigen Abständen erscheinen neue Versionen von Programmiersprachen, Entwicklungsumgebungen und Betriebssystemen. Manche dieser Updates verbessern lediglich die Stabilität oder ergänzen praktische neue Funktionen. Andere wiederum ersetzen bekannte Mechanismen durch neue. Entsprechend wichtig ist es, als Entwickler up to date zu sein.

Erfreulicherweise gibt es diverse Ressourcen, über die Sie sich auf dem Laufenden halten können. Nachfolgend möchte ich Ihnen einige davon vorstellen, die ich selbst als regelmäßige Anlaufstellen nutze.

1.6.1 Apple-Developer-App

Für macOS, iOS und tvOS stellt Apple eine kostenlose Developer-App zur Verfügung (siehe Bild 1.5). Darin finden Sie eine Vielzahl an Videos und diverse Artikel, die spezifische Aspekte der App-Entwicklung erläutern.

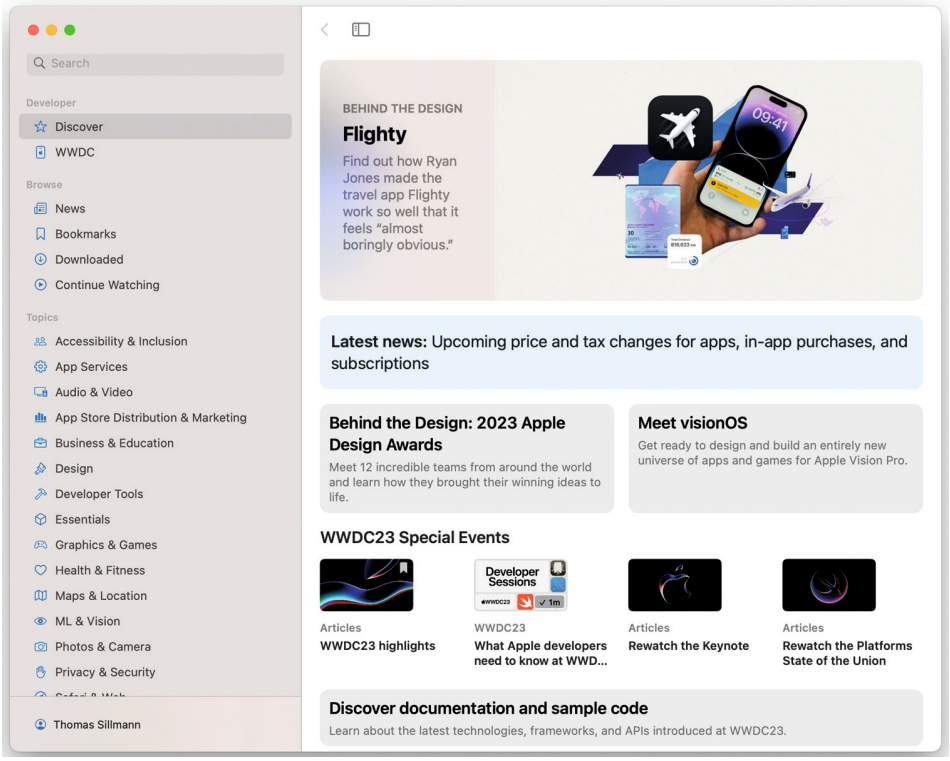

Bild 1.5 Apples offizielle Developer-App ist die ideale Ergänzung für alle Entwickler.

Insbesondere erhalten Sie über die App Zugriff auf die verschiedenen Session-Videos, die Apple im Zuge seiner alljährlichen Entwicklerkonferenz (Worldwide Developers Conference, kurz WWDC) veröffentlicht. Eine bessere Möglichkeit, sich über die neuesten Techniken im Bereich Swift, iOS und Co. zu informieren, gibt es wohl nicht.

1.6.2 Apples Developer-Website

Eine zentrale Anlaufstelle für alle Apple-Entwickler stellt Apples offizielle Developer-Website dar, die Sie mittels des Links *https://developer.apple.com* erreichen (siehe Bild 1.6). Sie liefert Ihnen eine Übersicht über die neuesten Entwicklungen und ermöglicht

den Zugriff auf Beta-Versionen von iOS und Co. Über diese Website erstellen Sie zudem Ihren eigenen Entwickler-Account und können auf verschiedene Ressourcen wie Zertifikate und App-IDs zurückgreifen. Mehr zu diesen Möglichkeiten erfahren Sie im letzten Teil dieses Buches.

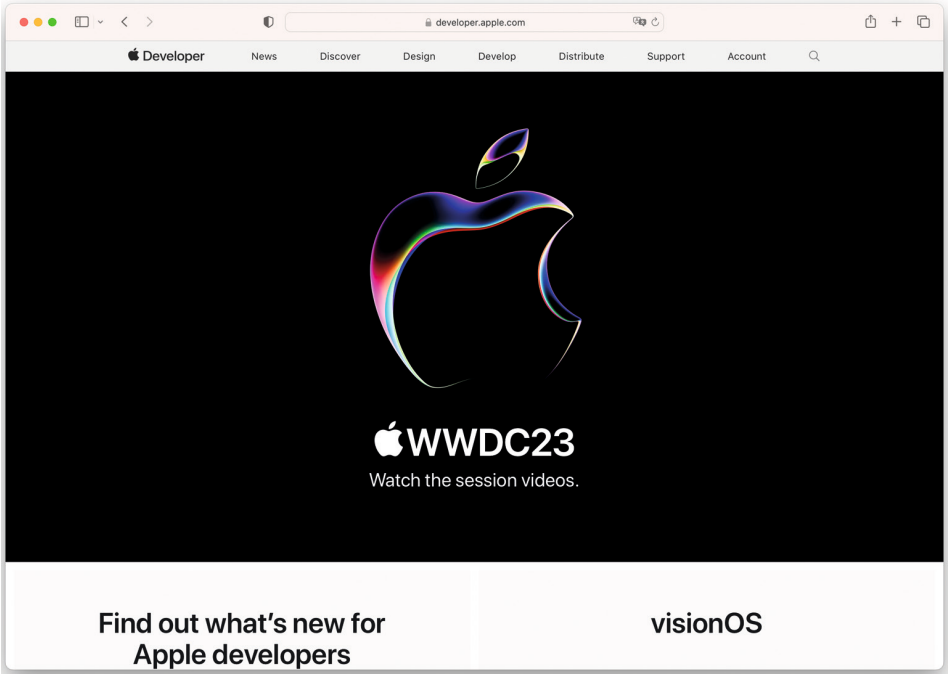

Bild 1.6 Über Apples offizielle Entwickler-Website haben Sie Zugriff auf aktuelle Infos und Beta-Versionen.

1.6.3 Swift.org

Wenn Sie sich im Speziellen für die Programmiersprache Swift interessieren, ist die offizielle Website *swift.org* genau das richtige (siehe Bild 1.7). Neben grundlegenden Informationen zu Swift finden Sie dort auch eine vollständige Dokumentation sowie einen Blog, der über kommende Updates und Änderungen der Sprache informiert. Vorbeischauen lohnt sich!

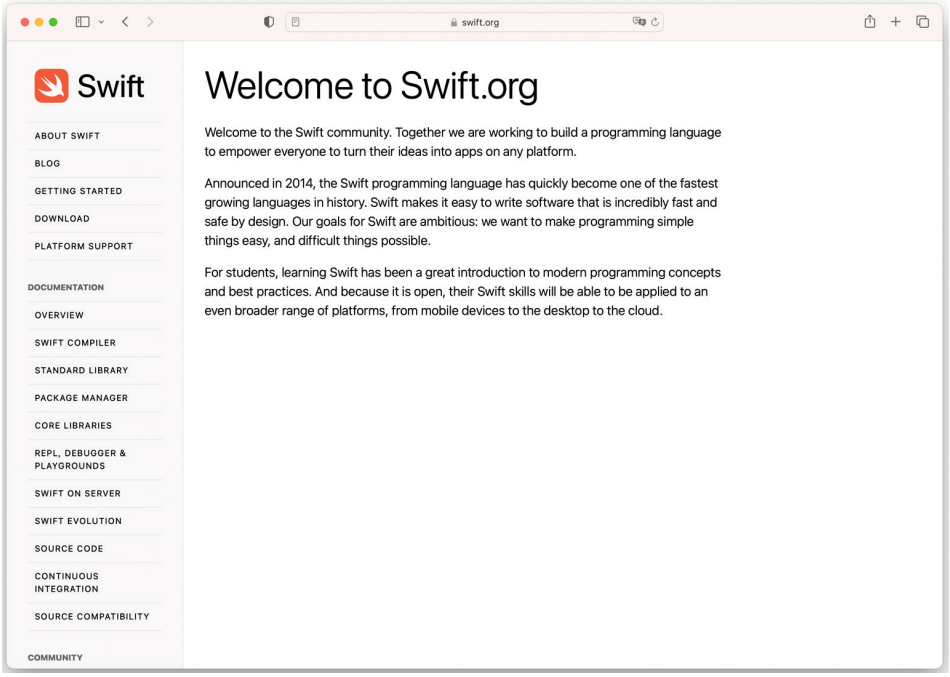

Bild 1.7 Die geballte Ladung Swift gibt es auf der offiziellen Website der Sprache unter *swift.org*.

1.6.4 In eigener Sache

Sie finden unter

https://letscode.thomassillmann.de

meinen ganz persönlichen Entwickler-Blog. Dort veröffentliche ich in unregelmäßigen Abständen neue Beiträge rund um die App-Entwicklung für Apple-Plattformen. Außerdem gibt es auf meinem YouTube-Kanal unter

https://www.youtube.com/user/Sillivan1988

regelmäßig neue Videos zur App-Entwicklung.

2 Grundlagen der Programmierung

In diesem Kapitel möchte ich Ihnen eine Einführung in die Grundlagen der Programmierung mit Swift geben. Es gibt Ihnen einen ersten Einblick in die Swift Standard Library, zeigt das Erstellen und Verwenden von Variablen und Konstanten und wie Sie Ihren Quellcode mithilfe von Kommentaren dokumentieren. Wenn Sie dabei sind, Swift zu lernen, empfehle ich Ihnen, die Beispiele dieses Buches in einem Playground auszuprobieren, um so möglichst schnell ein Gefühl für die Sprache zu bekommen und aktiv Code zu schreiben.

2.1 Grundlegendes

Im Folgenden stelle ich Ihnen verschiedene Bestandteile und Funktionen von Swift vor, die die Basis für die Programmierung darstellen.

2.1.1 Swift Standard Library

Die Swift Standard Library enthält ein umfangreiches Set an verschiedensten Klassen und Funktionen (siehe Bild 2.1). Sie ist Teil der Programmiersprache Swift, sodass alles, was Teil der Standard Library ist, auch in jedem Swift-Programm verwendet werden kann.

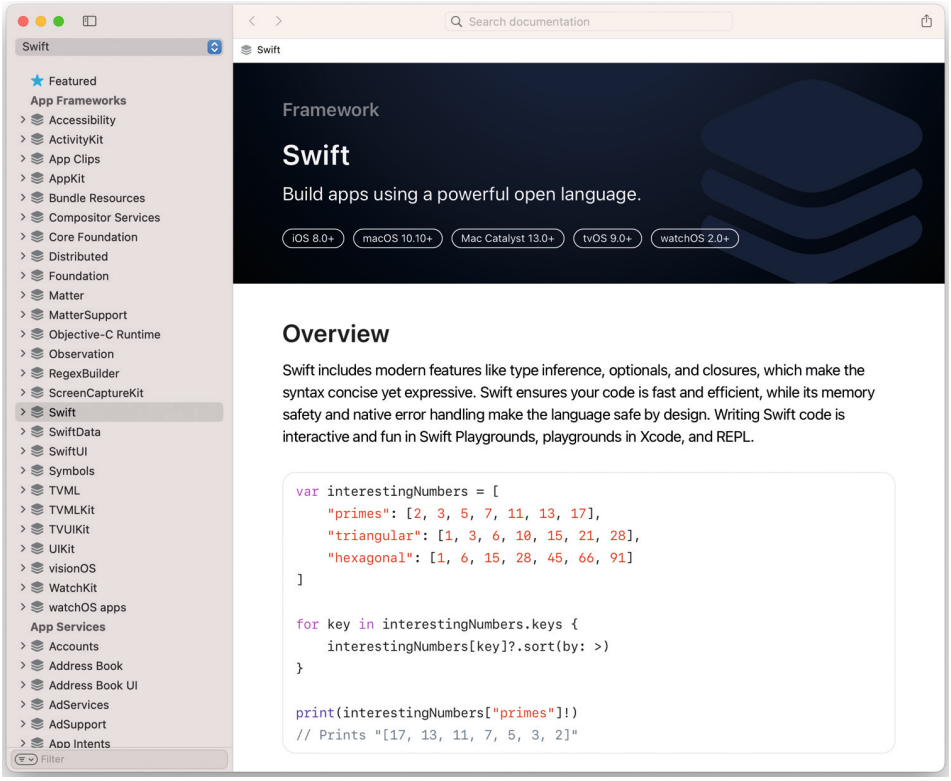

Bild 2.1 Die Swift Standard Library enthält ein umfangreiches Set an Funktionen, die uns bei der Programmierung mit Swift immer zur Verfügung stehen.

Dabei werden wir vielen sogenannten *Typen* der Swift Standard Library begegnen (was ein Typ genau ist und wie man selbst welche deklariert, folgt im Laufe dieses Kapitels). Dazu gehören beispielsweise die Typen `Int`, `Double`, `Character`, `String`, `Array` oder `Dictionary`. Die folgende Tabelle 2.1 gibt einen kurzen Überblick über einige der wichtigsten und grundlegendsten Typen für die Programmierung mit Swift, an passender Stelle im Buch werden diese auch noch tiefergehend beschrieben.

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library

Fundamental Type	Beschreibung	Beispiele
`Int`	Ein Integer (`Int`) stellt eine Ganzzahl dar.	`19` `99`
`Float`	Bei `Float` handelt es sich um eine Fließkommazahl	`19.99` `49.94`

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library *(Fortsetzung)*

Fundamental Type	Beschreibung	Beispiele
`Double`	Auch bei `Double` handelt es sich um eine Fließkommazahl, allerdings ist der Wertebereich von `Double` deutlich größer als der von `Float`; entsprechend belegt ein Double auch mehr Speicherplatz im System als ein `Float`.	`99.19` `94.49`
`Bool`	Bei `Bool` handelt es sich um einen sogenannten Wahrheitswert, dieser kann somit entweder wahr oder falsch (`true` oder `false`) sein.	`true` `false`
`String`	Ein `String` repräsentiert eine Zeichenkette.	`"Mein Name ist Thomas Sillmann."`
`Array`	In einem `Array` können mehrere Werte und Objekte abgelegt werden. Das `Array` erlaubt dann den Zugriff auf die Werte und Objekte, die es hält. Ein `Array` kann dabei beliebige Typen von Werten und Objekten beinhalten.	`["Erster Wert des Arrays", "Zweiter Wert des Arrays"]`
`Dictionary`	Ein `Dictionary` hält mehrere Werte und Objekte, ähnlich wie ein `Array`, allerdings ist jeder Wert und jedes Objekt einem einzigartigen Schlüssel innerhalb des Dictionaries zugeordnet. Anhand dieses Schlüssels können dann gezielt Werte ausgelesen, abgefragt und verändert werden.	`["Schlüssel 1": "Wert für Schlüssel 1", "Schlüssel 2": "Wert für Schlüssel 2"]`

Sie müssen zum jetzigen Zeitpunkt noch nicht mehr über die genannten Typen wissen, weitere Informationen zu ihnen folgen im Laufe dieses Buches an passender Stelle.

2.1.2 print

Im Laufe dieses Buches werden Sie sehr viele Elemente und Funktionen der Swift Standard Library kennenlernen. Eine der von mir am häufigsten verwendeten Befehle nennt sich `print(_:separator:terminator:)` und dient dazu, Text in der Konsole auszugeben. Ein Beispiel zeigt Listing 2.1. Wo immer diese Funktion zum Einsatz kommt, werde ich in den zugehörigen Listings auch die jeweilige Ausgabe (oder im

Falle mehrere Befehle auch alle jeweiligen Ausgaben) am Ende als Kommentar mit aufführen.

Listing 2.1 Einfache Konsolenausgabe mittels `print`

```
print("Das ist eine Konsolenausgabe")
// Das ist eine Konsolenausgabe
```

Darüber hinaus werde ich der Einfachheit halber, wo immer diese Funktion verwendet wird, auf diese im Fließtext mit `print` verweisen und mir die eigentlich korrekte Bezeichnung aus Platzgründen sparen.

2.1.3 Befehle und Semikolons

Bei der Entwicklung mit Swift schreibt man verschiedene aufeinanderfolgende Befehle, um damit am Ende ein funktionsfähiges Programm umzusetzen. Pro Zeile wird dabei genau ein Befehl geschrieben, beispielsweise um eine Variable zu erstellen oder einen Text auf der Konsole auszugeben. Jeder neue Befehl folgt in einer neuen Zeile (siehe Listing 2.2).

Listing 2.2 Schreiben eines Befehls pro Zeile

```
print("Das ist ein erster Befehl.")
print("Anschließend folgt ein zweiter.")
print("Und zum Abschluss ...")
print("... noch ein vierter!")
```

In vielen anderen Programmiersprachen muss jeder Befehl mit einem Semikolon (`;`) abgeschlossen werden. In Swift ist das ebenfalls möglich, aber kein Muss (wie das Listing von eben gezeigt hat). Sie können den Code aus Listing 2.2 also auch so, wie in Listing 2.3 gezeigt, umsetzen und am Ende eines jeden Befehls ein Semikolon setzen.

Listing 2.3 Schreiben eines Befehls mit abschließendem optionalen Semikolon

```
print("Das ist ein erster Befehl.");
print("Anschließend folgt ein zweiter.");
print("Und zum Abschluss...");
print("...noch ein vierter!");
```

Ein Semikolon zum Abschluss ist nur dann Pflicht, wenn man *mehrere* Befehle in einer Zeile schreiben möchte (siehe Listing 2.4).

Listing 2.4 Schreiben mehrerer Befehle in einer einzigen Zeile

```
print("Erster Befehl ..."); print("... direkt gefolgt vom zweiten!")
```

Der letzte Befehl in der Zeile muss wiederum nicht zwingend mit einem Semikolon abgeschlossen werden.

Semikolon – ja oder nein?

Womöglich fragen Sie sich nach diesem Abschnitt, was nun die bessere Lösung ist; Befehle mit einem Semikolon abzuschließen oder nicht? Und sollten in Swift mehrere Befehle in eine einzige Zeile geschrieben werden?

Ob und wie Sie letztlich das Semikolon in Swift auf die gezeigte Art und Weise verwenden, ist zunächst einmal voll und ganz Ihnen überlassen. Ich allerdings orientiere mich bei der Arbeit mit Swift an Apples Vorgehen aus der offiziellen Dokumentation, und dort wird prinzipiell **kein** Semikolon bei der Programmierung mit Swift eingesetzt (auch mehrere Befehle pro Zeile finden sich dort nicht). Wenn Sie also nicht gerade ein extremer Fan von Semikolons sind, dann würde ich Ihnen empfehlen, es genauso zu handhaben und einen Befehl pro Zeile zu schreiben – ohne abschließendes Semikolon.

2.1.4 Operatoren

Operatoren dienen dazu, im Code Befehle (wie beispielsweise Zuweisungen oder Berechnungen) durchzuführen. Da sich Operatoren durch viele Bereiche der Programmiersprache ziehen, möchte ich Ihnen gleich an dieser Stelle eine Übersicht der in Swift verfügbaren Operatoren geben (siehe Tabelle 2.2). An den Stellen im Buch, an denen diese Operatoren konkret zum Einsatz kommen, erhalten Sie weitere Erläuterungen und Ergänzungen dazu.

Tabelle 2.2 Operatoren in Swift

Operator	Art	Funktion
=	Zuweisungs-operator	Weist den Wert auf der rechten Seite des Operators dem Objekt auf der linken Seite zu.
==	Vergleichs-operator	Prüft, ob der Wert links vom Operator mit dem rechts vom Operator identisch ist.
!=	Vergleichs-operator	Prüft, ob der Wert links vom Operator mit dem rechts vom Operator nicht identisch ist.
<	Vergleichs-operator	Prüft, ob der Wert links vom Operator kleiner dem rechts vom Operator ist.
<=	Vergleichs-operator	Prüft, ob der Wert links vom Operator kleiner oder gleich dem rechts vom Operator ist.

Operator	Art	Funktion
>	Vergleichs-operator	Prüft, ob der Wert links vom Operator größer dem rechts vom Operator ist.
>=	Vergleichs-operator	Prüft, ob der Wert links vom Operator größer oder gleich dem rechts vom Operator ist.
+	Berechnungs-operator	Dient zur Durchführung von Additionen.
-	Berechnungs-operator	Dient zur Durchführung von Subtraktionen.
*	Berechnungs-operator	Dient zur Durchführung von Multiplikationen.
/	Berechnungs-operator	Dient zur Durchführung von Divisionen.
%	Berechnungs-operator	Dient zur Berechnung des Rests bei einer Division.
+=	Berechnungs-operator	Erhöht den Wert links vom Operator um den Wert rechts vom Operator.
-=	Berechnungs-operator	Verringert den Wert links vom Operator um den Wert rechts vom Operator.
&&	Logischer Operator	Verknüpft zwei Bedingungen mittels UND; ist eine von ihnen `false`, ist auch das Ergebnis `false`.
||	Logischer Operator	Verknüpft zwei Bedingungen mittels ODER; ist eine von beiden `true`, ist auch das Ergebnis `true`.
!	Logischer Operator	Kehrt einen Wahrheitswert um (`true` wird `false`, `false` wird `true`).
...	Range-Operator	Erstellt eine Wertereihe, die mit dem Wert links vom Operator beginnt und mit einschließlich dem Wert rechts vom Operator endet. Dabei darf der Wert links vom Operator nicht größer sein als der Wert rechts vom Operator.
..<	Range-Operator	Erstellt eine Wertereihe, die mit dem Wert links vom Operator beginnt und mit ausschließlich dem Wert rechts vom Operator endet. Dabei darf der Wert links vom Operator nicht größer sein als der Wert rechts vom Operator.
??	Nil-Operator	Prüft den optionalen Wert links vom Operator. Ist dieser `nil`, wird der Wert rechts vom Operator zurückgegeben, andernfalls wird der Wert links entpackt und zurückgegeben.

2.2 Variablen und Konstanten

Mithilfe von Variablen und Konstanten speichern Sie Werte zwischen, die Sie dann auslesen und weiterverarbeiten können. Einer Konstanten kann nur einmalig ein Wert zugewiesen werden, dieser ist anschließend nicht mehr veränderbar. Der Versuch, den Wert einer Konstanten anschließend zu ändern, endet in einem Compiler-Fehler. Im Gegensatz dazu kann der einer Variablen zugewiesene Wert jederzeit geändert werden.

2.2.1 Erstellen von Variablen und Konstanten

Eine Variable wird in Swift mittels des Schlüsselworts `var` deklariert, eine Konstante mittels `let`. Nach dem jeweiligen Schlüsselwort folgt der gewünschte Name für die Variable beziehungsweise Konstante. Dieser beginnt in Swift typischerweise mit einem Kleinbuchstaben. Setzt sich der Name aus mehreren verschiedenen Wörtern zusammen, so beginnt man jedes folgende Wort typischerweise mit einem Großbuchstaben.

Listing 2.5 zeigt ein Beispiel dazu. Dort wird eine Variable und eine Konstante deklariert und dieser direkt ein Wert (in diesem Fall ein String) zugewiesen. Die Zuweisung erfolgt mithilfe des Zuweisungsoperators `=`.

Listing 2.5 Erstellen von Variablen und Konstanten

```
var aVariable = "Eine Variable"
let aConstant = "Eine Konstante"
```

Um nach der Deklaration auf die Werte von Variablen und Konstanten zuzugreifen, nutzt man einfach den vergebenen Variablen- beziehungsweise Konstantennamen. So wird in Listing 2.6 auf die zuvor erstellte Variable `aVariable` zugegriffen und ihr ein neuer Wert zugewiesen.

Listing 2.6 Zugriff auf eine erstellte Variable

```
aVariable = "Ein neuer String"
```

Die Zuweisung eines Werts zu einer Variablen würde bei der zuvor deklarierten Konstanten `aConstant` nicht funktionieren, da Konstanten wie beschrieben nur einmalig ein Wert zugewiesen werden kann und dieser anschließend unveränderlich ist. Ein Versuch, den Wert einer Konstanten im Nachhinein zu ändern, führt immer zu einem Compiler-Fehler (siehe Listing 2.7).

Listing 2.7 Fehler beim Versuch des Änderns einer Konstanten

```
aConstant = "Eine neue Konstante"
// Compiler-Fehler: aConstant kann nicht verändert werden.
```

Wann Variable, wann Konstante?

Möglicherweise denken Sie nach dem Lesen dieses Abschnitts, dass es sinnvoll ist, sicherheitshalber lieber immer eine Variable statt eine Konstante zu erstellen, da Sie diese im Zweifelsfall noch verändern können. Das sollten Sie aber per se keinesfalls tun.

Denn diese Medaille hat noch eine zweite Seite: Sobald Sie beispielsweise einen neuen Wert erstellen, der innerhalb Ihres Programms unveränderlich sein soll (beispielsweise, weil er eine grundlegende und essenzielle Information enthält), dann können Sie genau dieses gewünschte Verhalten damit sicherstellen, diesen Wert mittels `let` als Konstante zu deklarieren. Wenn Sie dann fälschlicherweise an einer Stelle in Ihrem Projekt nun doch versuchen, genau diesen Wert zu ändern, dann macht Sie der Compiler direkt auf dieses Problem aufmerksam. Und genau für solche Zwecke – für Werte, die einmal gesetzt und anschließend nicht mehr verändert werden sollen – sind Konstanten da.

Das geht sogar so weit, dass in Swift generell der Grundsatz gilt: Wenn ein Wert nicht geändert werden muss oder soll, dann deklariere ihn als Konstante! Erstellen Sie daher im Zweifelsfall lieber eine unveränderliche Konstante als eine Variable. Sollte sich das später doch als möglicher Fehler herausstellen, ist es immer noch ein Leichtes, die Deklaration von einer Konstanten hin zu einer Variablen zu verändern.

2.2.2 Variablen und Konstanten in der Konsole ausgeben

Um den Wert von Variablen und Konstanten auf der Konsole auszugeben (beispielsweise bei der Suche nach Fehlern im Code) steht in Swift die Funktion `print` zur Verfügung. Typischerweise wird `print` ein String übergeben, der anschließend in der Konsole ausgegeben wird (siehe dazu auch den vorherigen Abschnitt 2.1.2, „print“). Sie können innerhalb dieses Strings aber auch eine Variable oder Konstante als eine Art Platzhalter übergeben, deren Wert dann in den String der `print`-Funktion eingefügt und ausgegeben wird. Um eine Variable oder Konstante auf die genannte Art und Weise in einen String einzubinden, müssen Sie sie innerhalb des Strings besonders kennzeichnen. Dazu nutzen Sie den folgenden Code:

```
\(<VARIABLE ODER KONSTANTE>)
```

In Listing 2.8 sehen Sie einmal ein Beispiel dazu, wie die Werte von Variablen und Konstanten mittels `print` ausgegeben werden können. Dazu werden die im vorherigen Abschnitt erstellte Variable `aVariable` und die Konstante `aConstant` verwendet.

Listing 2.8 Ausgabe der Werte von Variablen und Konstanten mittels `print`

```
print("aVariable hat folgenden Wert: \(aVariable)")
print("aConstant hat folgenden Wert: \(aConstant)")
// aVariable hat folgenden Wert: Ein neuer String
// aConstant hat folgenden Wert: Eine neue Konstante
```

Das gezeigte Vorgehen wird auch als *String Interpolation* bezeichnet; mehr dazu erfahren Sie in Kapitel 4, „Typen in Swift“.

2.2.3 Type Annotation und Type Inference

Variablen und Konstanten in Swift sind immer einem ganz bestimmten Typ zugeordnet. Eine Variable ist beispielsweise also entweder eine Zahl *oder* ein String. Handelt es sich bei ihr um eine Zahl, dann können ihr auch nur Zahlen und keine Strings zugewiesen werden, umgekehrt gilt genau das Gleiche. Dieses Verhalten wird als *Typsicherheit* bezeichnet, da man sich darauf verlassen kann, dass eine Variable oder Konstante immer nur einen Wert passend zu ihrem Typ besitzt.

Wenn Sie eine neue Variable oder Konstante erstellen, können Sie direkt angeben, von welchem Typ diese Variable beziehungsweise Konstante ist. Dazu fügen Sie nach dem Namen der Variablen oder Konstanten einen Doppelpunkt, gefolgt vom gewünschten Typ, ein. In Listing 2.9 sehen Sie ein Beispiel dazu.

Listing 2.9 Typzuweisung beim Erstellen von Variablen und Konstanten

```
var aString: String
let anInteger: Int
```

Hier wird festgelegt, dass die Variable `aString` vom Typ `String` ist und die Konstante `anIntenger` vom Typ `Int` (sowohl bei `String` als auch bei `Int` handelt es sich um automatisch bei der Programmierung mit Swift zur Verfügung stehende Typen aus der Swift Standard Library). Möchte man diesen beiden nun einen Wert zuweisen, so ist darauf zu achten, dass `aString` nur eine Zeichenkette entgegennehmen kann, während man `anInteger` nur eine Ganzzahl zuweisen kann (siehe Listing 2.10). Der Versuch, ihnen einen Wert eines anderen Typs zuzuweisen, hätte einen Compiler-Fehler zur Folge.

Listing 2.10 Wertzuweisung passend zu den Typen von Variablen und Konstanten

```
aString = "Ein mittels Type Annotation erstellter String"
anInteger = 19
```

Das gezeigte Vorgehen der direkten Typzuweisung beim Erstellen einer Variablen oder Konstanten wird als *Type Annotation* bezeichnet. Sollte diese nicht angewendet werden und – wie in den vorherigen Listings dieses Abschnitts zu sehen war – einer neuen Variablen oder Konstanten stattdessen direkt ein Wert zugewiesen werden, dann tritt die sogenannte *Type Inference* in Kraft. Fehlt nämlich eine konkrete Typzuweisung mittels Type Annotation, dann ermittelt Swift selbst, welchen Typ die Variable oder Konstante besitzen soll, sobald ihr ein Wert zugewiesen wird. Betrachten wir dazu einmal in Listing 2.11 die Erstellung einer neuen Konstanten und Variablen mittels Type Inference.

Listing 2.11 Erstellen neuer Variablen mittels Type Inference

```
let myName = "Thomas Sillmann"
var myAge = 28
// myName ist vom Typ String
// myAge ist vom Typ Int
```

Auch wenn es im Listing selbst nicht explizit angegeben ist, legt Swift automatisch sowohl für die Konstante `myName` als auch für die Variable `myAge` einen Typ fest, ausgehend von dem zugewiesenen Wert. So entspricht `myName` nun dem Typ `String` und `myAge` dem Typ `Int`.

Wann sollten Sie nun welches der beiden Verfahren einsetzen? Wann ist die explizite Typzuweisung mittels Type Annotation notwendig und in welchen Fällen kann man Swift den Typ selbst mittels Type Inference ermitteln lassen?

Generell ist der Einsatz von Type Annotation in zwei Situation zwingend notwendig:

- Wenn Sie einer neuen Variablen oder Konstanten bei deren Deklaration noch keinen Wert zuweisen, müssen Sie in jedem Fall den gewünschten Typ für die Variable oder Konstante angeben (so wie in Listing 2.9); andernfalls kommt es zu einem Compiler-Fehler.
- Wenn der mittels Type Inference von Swift ermittelte Typ bei der Erstellung einer Variablen oder Konstanten nicht dem gewünschten Typ entspricht, muss ebenfalls explizit der korrekte Typ mittels Type Annotation angegeben werden.

Den zweiten Punkt möchte ich zum besseren Verständnis noch einmal anhand eines Beispiels erläutern. Dazu wird in Listing 2.12 eine neue Variable namens `aDouble` erstellt und ihr der Zahlenwert `99` zugewiesen. Wie der Name der Variablen andeutet, soll diese im Code als `Double` (also als Fließkommazahl) verwendet werden können.

Listing 2.12 Erstellen einer neuen Variablen mit dem gewünschten Typ `Double`

```
Var aDouble = 99
// aDouble entspricht dem Typ Int
```

Zwar ist der gezeigte Code korrekt, allerdings handelt es sich bei `aDouble` nun nicht um eine Variable vom gewünschten Typ `Double`, sondern um eine vom Typ `Int`. Denn

Swift vermutet hinter der zugewiesenen Ganzzahl `99` nun einmal keine Fließkommazahl, auch wenn `99` natürlich nichtsdestoweniger ein valider Wert für eine Fließkommazahl wäre. Der Versuch, `aDouble` nun im Nachhinein einen Wert wie `19.99` zuzuweisen, würde ebenfalls in einem Compiler-Fehler enden. Daher ist es in so einem Fall zwingend notwendig, den gewünschten Typ ebenfalls explizit mittels Type Annotation anzugeben, wie in Listing 2.13 zu sehen.

Listing 2.13 Erstellen einer neuen `Double`-Variablen mittels Type Annotation

```
var aDouble: Double = 99
```

Damit ist trotz der Zuweisung einer Ganzzahl die Variable `aDouble` vom Typ `Double` und sie kann somit auch mit Fließkommazahlen umgehen.

2.2.4 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

Sie haben in Swift die Möglichkeit, mehrere Variablen und Konstanten direkt in einem Befehl zu erstellen und ihnen dabei optional bereits Werte zuzuweisen. Dazu beginnen Sie den entsprechenden Befehl entweder mit dem Schlüsselwort `var` (für zu erstellende Variablen) oder `let` (für zu erstellende Konstanten) und benennen dann kommasepariert alle neu zu erstellenden Variablen beziehungsweise Konstanten. Dabei können Sie entweder allen oder einzelnen Elementen direkt nach dem Namen auf die bekannte Art und Weise einen Wert zuweisen oder einen festen Typ mittels Type Annotation definieren. In Listing 2.14 sehen Sie einige Beispiele dazu, wie dieses Prinzip praktisch angewendet werden kann.

Listing 2.14 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

```
var firstValue: Int, secondValue: Double, thirdValue: String
var firstString, secondString, thirdString: String
let firstInt = 19, secondInt = 99
let numericValue = 19, numericString = "99"
```

Besonders interessant ist dabei auch die zweite Zeile `var firstString, secondString, thirdString: String`, in der nur eine einzige Type Annotation ganz am Ende erfolgt. Dadurch wird allen in diesem Befehl neu erstellten Variablen der am Ende explizit definierte Typ `String` zugewiesen, womit man sich die wiederholende Schreibarbeit spart, möchte man mehrere neue Variablen oder Konstanten von ein und demselben Typ auf einmal definieren.

2.2.5 Namensrichtlinien

Bei der Benennung von Variablen und Konstanten in Swift haben Sie – gerade im Vergleich mit anderen Programmiersprachen – sehr viele Freiheiten. So können beispielsweise Sonderzeichen wie Pi π oder sogar Emojis für Variablen- und Konstantennamen verwendet werden (siehe Listing 2.15).

Listing 2.15 Verwendung von Sonderzeichen und Emojis als Variablen- und Konstantennamen

```
let π = 3.14159
let 🐸 = "Frog"
```

Dennoch sind einige Dinge nicht erlaubt und führen direkt zu einem Compiler-Fehler. Beispielsweise müssen Sie auf jegliche Leerzeichen in einem Variablen- oder Konstantennamen verzichten, ebenso wie auf mathematische Operatoren oder Pfeile. Auch dürfen Variablen- oder Konstantennamen nicht mit einer Ziffer beginnen, ansonsten sind Ziffern im Namen aber erlaubt.

Im Zweifel lieber drauf verzichten

So schön die genannten Möglichkeiten und Freiheiten bei der Benennung von Variablen und Konstanten auch sind, sollte man sich dennoch überlegen, ob und wann sie tatsächlich angebracht sind. Gerade Sonderzeichen und Emojis sind womöglich eher ungeeignet für den eigenen Code, auch wenn diese Möglichkeit – wie wir gesehen haben – in Swift ja durchaus zur Verfügung steht. Wenn es keinen konkreten oder sinnvollen Grund für die Verwendung dieser Sonderzeichen gibt, sollten Sie im Zweifelsfall lieber darauf verzichten und stattdessen mit den bekannten alphanumerischen Zeichen bei der Benennung von Variablen und Konstanten arbeiten.

2.3 Kommentare

Kommentare sind in der Programmierung ein beliebtes und zugleich sehr wichtiges Mittel zur Dokumentation des eigenen Quellcodes. Kommentare werden vom Compiler ignoriert und nicht ausgeführt, was bedeutet, dass alles, was Sie innerhalb von Kommentaren schreiben, keinen Einfluss auf die Funktionalität Ihrer Anwendung hat. Typischerweise geben Sie mit Kommentaren Aufschluss über die Funktionsweise bestimmter Befehle oder die Aufgabe von deklarierten Variablen und Konstanten.

In Swift gibt es zwei Arten von Kommentaren: solche, die genau für eine Zeile gelten und solche, die sich über beliebig viele Zeilen erstrecken.

Ein einfacher einzeiliger Kommentar wird mit zwei Slashs // eingeleitet, direkt im Anschluss beginnt der Kommentar. Alles, was also hinter den beiden Slashs steht, wird vom Compiler ignoriert und dient einzig und allein dazu, den Quellcode zu dokumentieren. In Listing 2.16 sehen Sie ein einfaches Beispiel dazu.

Listing 2.16 Ein einzeiliger Kommentar

```
// Ein Kommentar
```

Solch ein Kommentar kann sowohl am Anfang als auch am Ende einer Zeile stehen (am Ende bedeutet dabei nach dem letzten Befehl innerhalb dieser Zeile). Auch dazu sehen Sie ein kleines Beispiel in Listing 2.17.

Listing 2.17 Ein einzeiliger Kommentar nach einem Befehl

```
print("Hier wird noch Code ausgeführt ...") // ... dann folgt ein Kommentar!
```

Manchmal benötigt aber ein sinnvoller Kommentar mehr Platz als nur eine einzige Zeile, und hier kommen die mehrzeiligen Kommentare ins Spiel. Diese beginnen mit einem /* und enden mit einem */. Alles, was sich dazwischen – auch über mehrere Zeilen hinweg – befindet, gehört zum Kommentar (siehe Listing 2.18).

Listing 2.18 Ein mehrzeiliger Kommentar

```
/* Der Kommentar beginnt in der ersten Zeile ...
 ... erstreckt sich über die zweite ...
 ... und endet schließlich in der dritten! */
```

Dabei können mehrzeilige Kommentare in Swift sogar verschachtelt werden. Ein mehrzeiliger Kommentar kann also einen weiteren mehrzeiligen Kommentar enthalten. Wie so etwas aussehen kann, zeigt Listing 2.19.

Listing 2.19 Verschachtelte Kommentare

```
/* Hier beginnt der erste Kommentar ...
 /* ... und hier der zweite ...
 ... der in dieser Zeile bereits wieder endet ... */
 ... sowie auch abschließend der erste Kommentar. */
```

3 Schleifen und Abfragen

Abfragen und Schleifen erlauben die Ausführung von Code unter bestimmten Bedingungen. Nur wenn diese Bedingungen erfüllt sind, werden zugehörige Befehle ausgeführt. Schleifen kümmern sich dabei um das wiederholte Ausführen bestimmter Befehle, während mit Abfragen Bedingungen geprüft werden. In diesem Kapitel stelle ich Ihnen diese beiden Elemente im Detail vor und zeige, wie Sie sie in Swift verwenden können.

3.1 Schleifen

Mithilfe von Schleifen können ein oder mehrere Befehle mehrmals hintereinander wiederholt ausgeführt werden. Dabei bietet Swift verschiedene Techniken an, um derartige Schleifen umzusetzen. Welche das sind und welche Möglichkeiten Sie bieten, erfahren Sie in den folgenden Abschnitten.

3.1.1 For-In

Mithilfe einer `for-in`-Schleife führen Sie eine Reihe von Befehlen immer wieder für einen festgelegten Wertebereich durch. Bei diesem Wertebereich handelt es sich typischerweise um eine Range, die Sie einfach mithilfe der Range-Operatoren definieren können. Alternativ können auch Arrays und Dictionaries als Wertebereich definiert werden (dazu erfahren Sie später mehr in den entsprechenden Abschnitten).

Eine `for-in`-Schleife wird mithilfe des Schlüsselworts `for` eingeleitet, gefolgt von einem Platzhalter, den Sie frei wie eine Variable oder Konstante benennen können. Diesem

Platzhalter wird bei jedem Durchlauf der Schleife automatisch der jeweils aktuelle Wert aus dem festgelegten Wertebereich zugewiesen und er kann innerhalb der Schleife dazu verwendet werden, diesen Wert auszulesen und mit ihm zu arbeiten. Anschließend folgt das zweite Schlüsselwort `in` gefolgt vom eigentlichen Wertebereich, der für die Schleife gelten soll. Innerhalb von geschweiften Klammern wird anschließend der Code angegeben, der bei jedem einzelnen Schleifendurchlauf ausgeführt werden soll. Listing 3.1 zeigt einmal den grundlegenden Aufbau einer `for-in`-Schleife.

Listing 3.1 Grundlegender Aufbau von `for-in`

```
for <PLATZHALTER> in <WERTEBEREICH> {
    <AUSZUFÜHRENDER CODE PRO SCHLEIFENDURCHLAUF>
}
```

Ein einfaches Beispiel für solch eine `for-in`-Schleife sehen Sie in Listing 3.2. Dort wird ein Wertebereich von `1` bis einschließlich `10` angegeben und bei jedem Durchlauf der jeweils aktuelle Wert aus dem Wertebereich per `print` ausgegeben.

Listing 3.2 Durchlaufen einer Schleife mittels `for-in`

```
for currentValue in 1...10 {
    print("Durchlauf \(currentValue).")
}
// Durchlauf 1.
// Durchlauf 2.
// Durchlauf 3.
// Durchlauf 4.
// Durchlauf 5.
// Durchlauf 6.
// Durchlauf 7.
// Durchlauf 8.
// Durchlauf 9.
// Durchlauf 10.
```

Die Schleife wird für jeden Wert des Wertebereichs einmal durchlaufen, also für alle Zahlen von eins bis zehn. Dabei wird dem von uns definierten Platzhalter `currentValue` bei jedem Schleifendurchlauf der aktuelle Wert aus dem Wertebereich zugewiesen, sodass sich dieser dynamisch bei jedem Schleifendurchlauf verändert. Dabei ist zu beachten, dass sich der Platzhalter wie eine Konstante verhält, er kann also innerhalb der Schleife nicht geändert werden. Ebenso wenig steht der Platzhalter *außerhalb* der Schleife zur Verfügung; ein Zugriff auf `currentValue` in diesem Beispiel nach der geschlossenen geschweiften Klammer der `for-in`-Schleife ist somit nicht möglich.

In einigen Fällen ist der aktuelle Wert aus dem zu durchlaufenden Wertebereich einer `for-in`-Schleife innerhalb der Schleife selbst uninteressant (beispielsweise, weil man bestimmte Befehle einfach nur mehrmals hintereinander ausführen möchte, ohne dass dafür der jeweils aktuelle Wert aus dem Wertebereich notwendig wäre). In solchen Fällen kann der Platzhalter einfach durch einen Unterstrich (_) ersetzt wer-

den, womit innerhalb der Schleife nicht mehr auf den jeweiligen Wert des aktuellen Durchlaufs zugegriffen werden kann. Listing 3.3 zeigt ein Beispiel dazu.

Listing 3.3 Durchlaufen einer Schleife ohne Zugriff auf den aktuellen Wert des Wertebereichs

```
var multiplyValue = 19
let multiplier = 8
for _ in 0..<3 {
    multiplyValue *= multiplier
}
print("multiplyValue entspricht \(multiplyValue).")
// multiplyValue entspricht 9728.
```

Hier wird eine `for-in`-Schleife insgesamt dreimal durchlaufen (was über den Wertebereich festgelegt und definiert ist). Pro Durchlauf soll eine Variable `multiplyValue` mit dem Wert der Konstanten `multiplier` multipliziert und das Ergebnis wiederum in `multiplyValue` gespeichert werden. Der aktuelle Wert aus dem Wertebereich pro Schleifendurchlauf interessiert also nicht, weshalb für den Platzhalter lediglich ein _ eingesetzt wird.

„Einfache“ For-Schleife

In anderen Programmiersprachen (darunter auch in Objective-C) finden sich „einfache“ `for`-Schleifen, in denen eine Zählvariable, eine Bedingung für die Schleife sowie ein Intervall zur Manipulation der Zählvariablen nach jedem Schleifendurchlauf definiert werden. Solange die Bedingung erfüllt ist, wird die Schleife weiter durchlaufen, weshalb die Bedingung in der Regel an die Zählvariable gekoppelt ist, die nach jedem Schleifendurchlauf verändert wird.

Ein solches Konzept fehlt in Swift. War es in Version 1 der Programmiersprache noch vorhanden, ist es inzwischen vollumfänglich verschwunden, weshalb `for`-Schleifen nur noch mittels `for-in` umgesetzt werden können.

3.1.2 While

Eine `while`-Schleife enthält ein oder mehrere Befehle, die so lange wiederholt ausgeführt werden, wie eine festgelegte Bedingung erfüllt ist. Im Gegensatz zur zuvor vorgestellten `for-in`-Schleife ist `while` also nicht an einen festen Wertebereich, sondern stattdessen an eine Bedingung gekoppelt. Bei dieser Bedingung handelt es sich um einen booleschen Wert, der entweder `true` (Schleife wird durchlaufen) oder `false` (Schleife wird verlassen) sein kann. Es handelt sich bei dieser Bedingung also ent-

weder direkt um eine Variable vom Typ `Bool` oder um einen Vergleich von zwei Werten mithilfe von Vergleichsoperatoren.

Den grundlegenden Aufbau einer `while`-Schleife zeigt Listing 3.4.

Listing 3.4 Grundlegender Aufbau einer `while`-Schleife

```
while <BEDINGUNG> {
    <AUSZUFÜHRENDER CODE, SOLANGE BEDINGUNG ERFÜLLT IST>
}
```

Damit eine `while`-Schleife korrekt funktioniert, muss die zugrunde liegende Bedingung spätestens im Verlauf der Schleife irgendwann einen Status erreichen, in dem diese nicht mehr erfüllt ist. Andernfalls würde eine Endlosschleife entstehen, die letztlich zum Absturz des Programms führt. Ein einfaches Beispiel einer funktionierenden `while`-Schleife sehen Sie in Listing 3.5.

Listing 3.5 Durchlaufen einer `while`-Schleife mithilfe einer Zählvariablen

```
var index = 1
while index <= 10 {
    print("Durchlauf \(index).")
    index += 1
}
// Durchlauf 1.
// Durchlauf 2.
// Durchlauf 3.
// Durchlauf 4.
// Durchlauf 5.
// Durchlauf 6.
// Durchlauf 7.
// Durchlauf 8.
// Durchlauf 9.
// Durchlauf 10.
```

Basis der `while`-Schleife ist die Bedingung `index <= 10`. Die Schleife wird also nur ausgeführt, wenn die Variable `index` kleiner oder gleich 10 ist und so oft durchlaufen, wie diese Bedingung erfüllt ist. Aus diesem Grund ist auch der Befehl `index += 1` innerhalb der `while`-Schleife so immens wichtig. Würde dieser fehlen, würde sich der Wert der Variablen `index` niemals verändern und die gestellte Bedingung wäre ununterbrochen erfüllt, was dazu führt, dass die Schleife ohne Unterlass und ohne eine Chance auf Beendigung ausgeführt wird und das beschriebene Problem der sogenannten Endlosschleife entsteht. Denken Sie immer an diesen Aspekt, wenn Sie mit einer `while`-Schleife arbeiten.

3.1.3 Repeat-While

Die `repeat-while`-Schleife ist eine leicht abgewandelte Form der zuvor vorgestellten `while`-Schleife. Ebenso wie bei der `while`-Schleife ist das Durchlaufen einer `repeat-while`-Schleife an eine festgelegte Bedingung gekoppelt, allerdings mit dem Unterschied, dass der Code einer `repeat-while`-Schleife in jedem Fall wenigstens einmal durchlaufen wird, und das selbst dann, wenn die zugrunde liegende Bedingung der Schleife von Beginn an nicht erfüllt ist.

Listing 3.6 zeigt zunächst einmal den grundlegenden Aufbau einer `repeat-while`-Schleife in Swift.

Listing 3.6 Grundlegender Aufbau einer `repeat-while`-Schleife

```
repeat {
    <AUSZUFÜHRENDER CODE SOLANGE BEDINGUNG ERFÜLLT IST, MINDESTENS ABER EINMAL>
} while <BEDINGUNG>
```

Ein Beispiel für eine `repeat-while`-Schleife sehen Sie in Listing 3.7.

Listing 3.7 Durchlaufen einer `repeat-while`-Schleife mit einer Zählvariablen

```
var index = 1
repeat {
    print("Durchlauf \(index).")
    index += 1
} while index <= 10
// Durchlauf 1.
// Durchlauf 2.
// Durchlauf 3.
// Durchlauf 4.
// Durchlauf 5.
// Durchlauf 6.
// Durchlauf 7.
// Durchlauf 8.
// Durchlauf 9.
// Durchlauf 10.
```

Ein anderes Beispiel für eine `repeat-while`-Schleife zeigt Listing 3.8. Hier ist die gestellte Bedingung von Beginn an nicht erfüllt (was bei einer `while`-Schleife dafür sorgen würde, dass der Code innerhalb der Schleife niemals ausgeführt wird), dennoch wird der Code innerhalb von `repeat-while` – wie beschrieben – einmal ausgeführt.

Listing 3.8 Durchlauf einer `repeat-while`-Schleife selbst bei nicht erfüllter Bedingung

```
let shouldRepeatLoop = false
repeat {
    print("Schleifendurchlauf")
} while shouldRepeatLoop
// Schleifendurchlauf
```

3.2 Abfragen

Mithilfe von Abfragen können Sie festlegen, dass bestimmte Befehle nur unter bestimmten Bedingungen ausgeführt werden. Zur Umsetzung solcher Abfragen gibt es in Swift drei Techniken: `if`, `switch` und `guard`. Alle drei stelle ich Ihnen nun nacheinander im Detail vor.

3.2.1 If

Mithilfe des Schlüsselworts `if` erstellen Sie in Swift eine einfache Abfrage. `if` erwartet dabei eine Bedingung, die entweder *wahr* oder *falsch* sein kann; es handelt sich also um einen booleschen Wahrheitswert. Ist dieser wahr, wird der Code, der nach `if` innerhalb von geschweiften Klammern angegeben ist, ausgeführt, andernfalls nicht. Listing 3.9 zeigt den grundlegenden Aufbau einer `if`-Abfrage in Swift.

Listing 3.9 Grundlegender Aufbau einer `if`-Abfrage

```
if <BEDINGUNG> {
    <AUSZUFÜHRENDER CODE WENN BEDINGUNG WAHR>
}
```

Bedingungen müssen immer einen Boolean zurückliefern, der wie beschrieben auf `true` geprüft wird. Zu diesem Zweck können Sie entweder eine Variable oder eine Konstante vom Typ `Bool` als Bedingung anführen oder Vergleichs- und logische Operatoren nutzen, um daraus einen passenden Wahrheitswert zu generieren.

Keine runden Klammern um Bedingung notwendig

In den meisten anderen Programmiersprachen wird die zu prüfende Bedingung einer `if`-Abfrage innerhalb von runden Klammern deklariert. In Swift ist das nicht notwendig, aber dennoch möglich. Der in Listing 3.9 gezeigte Aufbau einer `if`-Abfrage kann also auch so wie in Listing 3.10 umgesetzt werden.

Listing 3.10 Grundlegender Aufbau einer `if`-Abfrage mit optionalen runden Klammern

```
if (<BEDINGUNG>) {
    <AUSZUFÜHRENDER CODE WENN BEDINGUNG WAHR>
}
```

Da in Swift generell auf die runden Klammern bei der Bedingung einer `if`-Abfrage verzichtet wird, werde ich auch im weiteren Verlauf des Buches keine runden Klammern um solche Bedingungen setzen.

Sollten Sie dieses Verfahren allerdings besser finden oder schlicht aus anderen Programmiersprachen gewohnt sein, spricht nichts dagegen, es auch in Swift anzuwenden.

In Listing 3.11 sehen Sie ein einfaches Beispiel für eine Abfrage. Dabei wird zunächst eine Variable vom Typ `String` erstellt und ihr ein Name zugewiesen. Anschließend wird in einer `if`-Abfrage dieser Name geprüft. Ist die Prüfung erfolgreich, wird eine Meldung auf der Konsole ausgegeben, andernfalls nicht.

Listing3.11 Abfrage eines Namens mittels `if`

```
let myName = "Thomas"
if myName == "Thomas" {
    print("Mein Name ist Thomas.")
}
// Mein Name ist Thomas.
```

Die Bedingung lautet in diesem Fall `myName == "Thomas"`. Ist diese wahr (was hier zutrifft), wird der Code innerhalb der geschweiften Klammern der `if`-Abfrage ausgeführt, andernfalls würde er ignoriert.

Darüber hinaus gibt es aber auch die Möglichkeit, eine `if`-Abfrage um einen weiteren Code-Block zu ergänzen: `else`. Der Code von `else` wird dann ausgeführt, wenn die Bedingung der `if`-Abfrage *nicht* wahr ist. Damit lässt sich eine Art Fallback umsetzen, um eine alternative Aktion auszuführen, wenn die zu prüfende Bedingung nicht erfüllt sein sollte. Dazu wird das Schlüsselwort `else` nach der geschlossenen geschweiften Klammer der `if`-Abfrage angeführt, gefolgt von einem weiteren geschweiften Klammernpaar, in dem sich dann der alternative Code befindet, der im Falle einer Nichterfüllung der Bedingung ausgeführt werden soll. Listing 3.12 zeigt den grundlegenden Aufbau einer `if`-Abfrage mit zusätzlichem `else`-Block.

Listing 3.12 Grundlegender Aufbau einer `if`-Abfrage mit `else`-Block

```
if <BEDINGUNG> {
    <AUSZUFÜHRENDER CODE WENN BEDINGUNG WAHR>
} else {
    <AUSZUFÜHRENDER CODE WENN BEDINGUNG NICHT WAHR>
}
```

In Listing 3.13 sehen Sie ein Beispiel einer solchen `if`-Abfrage, die erneut einen Namen prüft, dabei aber auch eine alternative Funktion bietet, sollte die Bedingung des Namensvergleichs nicht erfüllt sein.

Listing 3.13 Abfrage eines Namens mittels `if` und `else`

```
let anotherName = "Tobias"
if anotherName == "Thomas" {
    print("Mein Name ist Thomas.")
} else {
```

```
    print("Mein Name ist nicht Thomas.")
}
// Mein Name ist nicht Thomas.
```

Hier wird nun mittels der Bedingung `anotherName == "Thomas"` der Wert der Konstanten `anotherName` gegen den String `"Thomas"` geprüft. Da diese Bedingung hier nicht erfüllt ist (da `anotherName` den Wert `"Tobias"` besitzt), wird stattdessen der Code innerhalb des `else`-Blocks ausgeführt.

Ternary Conditional Operator

Der sogenannte Ternary Conditional Operator fungiert in Swift als Kurzschreibweise für eine If-Abfrage mit einem zusätzlichen `else`-Block. Er baut sich syntaktisch wie folgt auf:

```
<Bedingung> ? <Befehle wenn Bedingung true> : <Befehle wenn
Bedingung false>
```

Er eignet sich ideal, um auf Basis einer Bedingung einen von zwei Befehlen auszuführen oder einen von zwei Werten zurückzuliefern. Das Beispiel aus Listing 3.13 lässt sich mithilfe des Ternary Conditional Operators so, wie in Listing 3.14 zu sehen, umsetzen.

Listing 3.14 Einsatz des Ternary Conditional Operators

```
anotherName == "Thomas" ? print("Mein Name ist Thomas.") :
print("Mein Name ist nicht Thomas.")
```

Zu Beginn steht die zu prüfende Bedingung, gefolgt von einem Fragezeichen. Im Anschluss folgt der Befehl, der auszuführen ist, sollte die Bedingung `true` entsprechen. Nach einem Doppelpunkt ergänzen Sie dann noch den auszuführenden Befehl, sollte die Bedingung `false` zurückliefern.

Insbesondere bei der Erstellung von Nutzeroberflächen mit SwiftUI kommt der Ternary Conditional Operator des Öfteren zum Einsatz. Mehr zu diesem Thema erfahren Sie in Teil 3, „App-Entwicklung“.

Zu guter Letzt können Sie eine `if`-Abfrage aber nicht nur um einen alternativen `else`-Block ergänzen, sondern um beliebig viele sogenannte `else if`-Blöcke. Ein `else if`-Block verfügt dabei über eine weitere zu prüfende Bedingung sowie einen Satz an auszuführenden Befehlen, sollte die entsprechende Bedingung wahr sein. Das erlaubt es Ihnen, innerhalb einer `if`-Abfrage nicht nur eine konkrete Bedingung zu prüfen, sondern mehrere. Sobald eine Bedingung sich als wahr herausgestellt hat, wird der zugehörige Code ausgeführt und die `if`-Abfrage anschließend verlassen. Es wird also dann nicht noch geprüft, ob womöglich eine der nachfolgenden Bedingungen der `if`-Abfrage ebenfalls wahr ist.

`else if`-Blöcke werden immer *nach* der erstmaligen `if`-Bedingung und *vor* einem optional abschließenden `else`-Block definiert; ein `else if` kann also niemals nach einem `else`-Block erfolgen, ein solcher kennzeichnet immer das Ende einer `if`-Abfrage. In Listing 3.15 sehen Sie den grundlegenden Aufbau zur Verwendung von `else if`-Blöcken in einer `if`-Abfrage. Wie beschrieben können beliebig viele solcher Blöcke innerhalb einer `if`-Abfrage definiert werden.

Listing 3.15 Grundlegender Aufbau einer `if`-Abfrage mit `else if`- und `else`-Block

```
if <ERSTE BEDINGUNG> {
    <AUSZUFÜHRENDER CODE WENN ERSTE BEDINGUNG WAHR
} else if <ZWEITE BEDINGUNG> {
    <AUSZUFÜHRENDER CODE WENN ERSTE BEDINGUNG NICHT WAHR UND ZWEITE BEDINGUNG WAHR>
} else {
    <AUSZUFÜHRENDER CODE WENN ERSTE UND ZWEITE BEDINGUNG NICHT WAHR>
}
```

Ein Beispiel dazu sehen Sie in Listing 3.16. Es erweitert den Code aus Listing 3.13 um zwei `else if`-Blöcke.

Listing 3.16 Abfrage eines Namens mittels `if`, `else if` und `else`

```
if anotherName == "Thomas" {
    print("Mein Name ist Thomas.")
} else if anotherName == "Michaela" {
    print("Mein Name ist Michaela.")
} else if anotherName == "Tobias" {
    print("Mein Name ist Tobias.")
} else {
    print("Mein Name lautet anders.")
}
// Mein Name ist Tobias.
```

Wie beschrieben, ist der abschließende `else`-Block optional, weshalb er in diesem Fall auch gänzlich wegfallen kann; der Code würde noch immer wie gewünscht funktionieren (siehe Listing 3.17).

Listing 3.17 Verzicht auf optional abschließenden `else`-Block einer `if`-Abfrage

```
if anotherName == "Thomas" {
    print("Mein Name ist Thomas.")
} else if anotherName == "Michaela" {
    print("Mein Name ist Michaela.")
} else if anotherName == "Tobias" {
    print("Mein Name ist Tobias.")
}
// Mein Name ist Tobias.
```

Verknüpfen mehrerer Bedingungen

In manchen Fällen reicht es nicht aus, nur eine Bedingung auf ihre Richtigkeit zu prüfen, sondern mehrere. Nehmen wir an, Sie wollen mithilfe einer Abfrage überprüfen,

ob eine Zahl größer oder gleich 10, gleichzeitig aber kleiner als 100 ist. Dabei könnte ein Konstrukt, wie in Listing 3.18 gezeigt, entstehen.

Listing 3.18 Prüfen mehrerer Bedingungen

```
let number = 19
if number >= 10 {
    if number < 100 {
        print("number ist größer oder gleich 10 und kleiner als 100.")
    }
}
// number ist größer oder gleich 10 und kleiner als 100.
```

Der gezeigte Code ist zwar an sich korrekt und funktioniert, ist aber gleichzeitig sehr aufwendig. Kämen nun noch weitere Bedingungen hinzu, würde sich das Konstrukt immer weiter verschachteln und damit auch immer unübersichtlicher werden.

Aus diesem Grund haben Sie die Möglichkeit, mehrere Bedingungen bei einer `if`-Abfrage mithilfe der sogenannten *logischen Operatoren* miteinander zu verknüpfen. Dabei spielen die folgenden beiden eine essenzielle Rolle:

- UND-Operator `&&`: Nur wenn alle mittels UND-Operator verknüpften Bedingungen wahr sind, ist die gesamte Bedingung wahr. Ist auch nur eine Bedingung falsch, ist damit auch die gesamte Bedingung falsch.
- ODER-Operator `||`: Wenn eine der mittels ODER-Operator verknüpften Bedingungen wahr ist, ist die gesamte Bedingung wahr. Nur, wenn alle Bedingungen falsch sind, ist auch die gesamte Bedingung falsch.

In dem Beispiel aus Listing 3.18 haben wir es mit einer typischen UND-Verknüpfung zu tun: Nur, wenn der abgefragte Wert größer oder gleich zehn **und** kleiner als hundert ist, soll der zugehörige Code ausgeführt werden. Entsprechend können die beiden Bedingungen auch in einer einzigen `if`-Abfrage mittels `&&` zusammengenfasst werden, so wie in Listing 3.19 zu sehen.

Listing 3.19 Prüfen mehrerer Bedingungen mittels &&-Operator

```
if number >= 10 && number < 100 {
    print("number ist größer oder gleich 10 und kleiner als 100.")
}
// number ist größer oder gleich 10 und kleiner als 100.
```

Ein Beispiel für eine typische Abfrage mit ODER-Verknüpfung zeigt Listing 3.20. Hier wird der Code innerhalb des `if`-Blocks genau dann ausgeführt, wenn der Wert der Konstanten `number` **entweder** genau `19` **oder** genau `99` entspricht.

Listing 3.20 Prüfen mehrerer Bedingungen mittels ||-Operator

```
if number == 19 || number == 99 {
    print("number ist gleich 19 oder 99.")
}
// number ist gleich 19 oder 99.
```

Obwohl eine der Bedingungen in dieser Abfrage nicht erfüllt ist (`number` entspricht schließlich nicht 99), wird der zugehörige Code dennoch ausgeführt, da wenigstens eine andere Bedingung der ODER-Verknüpfung erfüllt ist.

Diese Form der Verknüpfung können Sie auch mischen und weiter verschachteln, wie in Listing 3.21 zu sehen. Die dort gezeigte Bedingung der `if`-Abfrage ist dann erfüllt, wenn `number` **entweder** größer oder gleich `0` **und** kleiner als `10` ist **oder** gleich `19` ist.

Listing 3.21 Verknüpfen mehrerer Bedingungen mit verschiedenen Operatoren

```
if number >= 0 && number < 10 || number == 19 {
    print("number ist entweder größer gleich 0 und kleiner als 10 oder gleich 19.")
}
// number ist entweder größer gleich 0 und kleiner als 10 oder gleich 19.
```

3.2.2 Switch

`switch` ist eine zweite Möglichkeit (neben dem zuvor vorgestellten `if`) zum Erstellen von Abfragen in Swift, dennoch unterscheidet es sich in Aufbau und Funktionsweise stark von `if`-Abfragen und bietet überdies deutlich mehr Möglichkeiten der Anwendung. Doch eins nach dem anderen.

Ein `switch`-Statement wird mit einer zu prüfenden Variablen oder Konstanten eingeleitet. Anschließend werden ein oder mehrere sogenannte *Cases* erstellt. Ein Case prüft die zuvor genannte Variable oder Konstante gegen einen oder mehrere definierte Werte. Entspricht sie einem dieser im Case definierten Werte, wird anschließend der zugehörige Code dieses Cases ausgeführt und die Abfrage anschließend verlassen. Den grundlegenden Aufbau einer einfachen `switch`-Abfrage sehen Sie in Listing 3.22.

Listing 3.22 Grundlegender Aufbau von `switch`

```
switch <VARIABLE ODER KONSTANTE> {
case <ZU VERGLEICHENDER WERT>:
    <AUSZUFÜHRENDER CODE WENN VARIABLE ODER KONSTANTE IDENTISCH MIT WERT>
default:
    <AUSZUFÜHRENDER CODE WENN VARIABLE ODER KONSTANTE KEINEM CASE ENTSPRICHT>
}
```

Die Anzahl der `case`-Blöcke einer `switch`-Abfrage ist variabel. Einer ist mindestens notwendig, darüber hinaus können beliebig viele weitere `case`-Blöcke vor dem abschließenden `default`-Block hinzugefügt werden, um andere Werte abzufragen. Dabei muss ein `case`-Block mindestens einen auszuführenden Befehl enthalten und darf niemals komplett leer sein.

Der `default`-Block ist in Swift speziell. Generell kann er mit dem `else`-Block bei einer `if`-Abfrage verglichen werden; so wird der darin deklarierte Code genau dann ausge-

führt, wenn die zu prüfende Variable oder Konstante keinem der Werte der vorherigen Cases entspricht. Im Gegensatz zu vielen anderen Programmiersprachen ist der `default`-Block in Swift aber nicht optional, sondern zwingend vorgeschrieben. Wann immer Sie also eine `switch`-Abfrage erstellen, müssen Sie auch einen `default`-Block anbieten, selbst wenn in dem Fall, dass keiner der deklarierten Cases zutrifft, nichts passieren soll. Einzige Ausnahme: Die Cases decken jeden möglichen Wert ab, den die Variable oder Konstante überhaupt annehmen kann; dann ist verständlicherweise ein `default`-Block nicht nötig (da er sowieso niemals aufgerufen würde) und er muss in diesem Fall auch weggelassen werden. Über diesen besonderen Fall erfahren Sie mehr in Kapitel 6, „Enumerations, Structures und Classes“.

In Listing 3.23 sehen Sie ein einfaches Beispiel für eine `switch`-Abfrage. Es wird eine zuvor deklarierte Konstante `name` gegen verschiedene Cases geprüft und anschließend der Code des passenden Case ausgeführt.

Listing 3.23 Abfrage eines Namens mittels `switch`

```
let name = "Michaela"
switch name {
case "Thomas":
    print("name entspricht Thomas.")
case "Michaela":
    print("name entspricht Michaela.")
case "Tobias":
    print("name entspricht Tobias.")
default:
    print("name entspricht einem anderen Wert.")
}
// name entspricht Michaela.
```

Eine `switch`-Abfrage prüft somit die übergebene Variable oder Konstante auf Gleichheit mit den verschiedenen Cases. Das ist zu vergleichen mit `if`-Abfragen, die eine Variable oder Konstante mithilfe des Vergleichsoperators `==` gegen eine andere Variable oder Konstante oder einen Wert prüfen.

Implicit und Explicit Fallthrough

Zu beachten ist, dass bei `switch` in Swift nur exakt der Code des zugehörigen Cases ausgeführt wird. Das ist deshalb so wichtig, da in vielen anderen Programmiersprachen (unter anderem in Objective-C) standardmäßig auch alle auf den passenden Case folgenden Cases mit ausgeführt werden, sofern dieses Verhalten nicht explizit verhindert wird. Eben dieses Verhalten – das Ausführen des passenden Cases sowie aller darauffolgenden – wird als *Implicit Fallthrough* bezeichnet. In Swift hingegen ist das Gegenteil der Fall, der *Explicit Fallthrough*. Das heißt, Sie können das genannte Verhalten auch in Swift nachbilden, müssen es aber eben *explizit* anstoßen.

Zu diesem Zweck dient das Schlüsselwort `fallthrough`. Sobald dieser Befehl innerhalb eines `case`-Blocks ausgeführt wird, wird dieser verlassen und der direkt darauffol-

gende case-Block ausgeführt – unabhängig davon, ob die zu vergleichenden Werte dieses zweiten case-Blocks denen der zu prüfenden Variablen und Konstanten entsprechen. In Listing 3.24 sehen Sie ein Beispiel dazu. Dafür wurde der Code aus Listing 3.23 in allen drei case-Blöcken um das Schlüsselwort fallthrough ergänzt.

Listing 3.24 Cases mit fallthrough

```
switch name {
case "Thomas":
    print("name entspricht Thomas.")
    fallthrough
case "Michaela":
    print("name entspricht Michaela.")
    fallthrough
case "Tobias":
    print("name entspricht Tobias.")
    fallthrough
default:
    print("name entspricht einem anderen Wert.")
}
// name entspricht Michaela.
// name entspricht Tobias.
// name entspricht einem anderen Wert.
```

In diesem Beispiel wird zunächst der passende Case für den Wert "Michaela" ausgeführt. Sobald darin das Schlüsselwort fallthrough erreicht wird, wird automatisch auch der Code des nächsten Cases ausgeführt. Da sich dort ebenfalls wieder das Schlüsselwort fallthrough findet, wird zu guter Letzt auch noch der Code des default-Blocks ausgeführt.

Verständlicherweise ist ein Verwenden von fallthrough innerhalb des default-Blocks verboten und führt umgehend zu einem Compiler-Fehler. Da nach dem default-Block niemals noch ein weiterer Block folgen kann, macht dort auch eine entsprechende Verwendung von fallthrough keinen Sinn.

Das Gegenteil zu fallthrough ist break. Dieses Schlüsselwort sorgt dafür, dass ein Case umgehend verlassen wird, ohne den darauffolgenden Block aufzurufen. Wie beschrieben, müssen Sie break in Swift nicht für das Ende eines case-Blocks verwenden, da dort die entsprechende Logik sowieso ausgeführt wird, auch ohne explizite Angabe von break.

Allerdings gibt es einen Sonderfall, in dem die Verwendung von break in Swift durchaus sinnvoll sein kann, nämlich dann, wenn Sie innerhalb des default-Blocks in switch keinen einzigen Befehl ausführen möchten. Schließlich ist der default-Block dennoch Pflicht und muss implementiert werden. Sie können in solchen Fällen also einfach innerhalb von default den Befehl break aufrufen, fertig. Listing 3.25 zeigt ein kleines Beispiel dazu.

Listing 3.25 Nutzen von `break` im `default`-Block

```
switch name {
case "Thomas":
    print("name entspricht Thomas.")
default:
    break
}
// name entspricht Thomas.
```

Compound Cases

Ein `case`-Block innerhalb einer `switch`-Abfrage kann die zu prüfende Variable beziehungsweise Konstante nicht nur mit einem, sondern sogar mit beliebig vielen verschiedenen Werten vergleichen; man spricht hierbei von den sogenannten *Compound Cases*, also Cases, die sich aus mehreren möglichen Werten zusammensetzen. Dazu werden die gewünschten Werte kommasepariert nach dem Schlüsselwort `case` und vor dem abschließenden Doppelpunkt nacheinander aufgeführt. Ein Beispiel dazu zeigt Listing 3.26.

Listing 3.26 Case mit mehreren möglichen Werten

```
switch name {
case "Thomas", "Michaela", "Tobias":
    print("name entspricht Thomas, Michaela oder Tobias.")
default:
    print("name entspricht einem anderen Wert.")
}
// name entspricht Thomas, Michaela oder Tobias.
```

Entspricht die zu überprüfende Variable oder Konstante einem der Werte eines `case`-Blocks, dann wird dieser entsprechend ausgeführt, so wie im gezeigten Fall.

Interval Matching

Sie können in `switch`-Cases Range-Operatoren nutzen, um damit schnell und einfach einen bestimmten Wertebereich für einen einzelnen Case abzufragen (anstatt alle Werte dieses Wertebereichs einzeln in einem Case kommasepariert voneinander aufzulisten). Dieses Verfahren wird auch als *Interval Matching* bezeichnet. Listing 3.27 zeigt ein konkretes Beispiel dazu.

Listing 3.27 `switch` mit Interval Matching

```
let value = 99
switch value {
case 0..<10:
    print("value ist einstellig.")
case 10..<100:
    print("value ist zweistellig.")
case 100..<1000:
    print("value ist dreistellig.")
```

```
default:
    break
}
// value ist zweistellig.
```

3.2.3 Guard

Mit `guard` erstellen Sie Abfragen, die umgekehrt zu den bereits vorgestellten `if`-Abfragen funktionieren. Wie bei `if` prüfen Sie auch bei `guard` eine Bedingung, führen anschließend aber innerhalb geschweifter Klammern den Code für den Fall aus, dass diese Bedingung *nicht* erfüllt ist; Sie starten also sozusagen mit dem `else`-Block einer `if`-Abfrage.

Doch das ist nicht die einzige Besonderheit von `guard`. Nach dem `else`-Block geht es direkt weiter mit dem Code, der ausgeführt werden soll, wenn die gestellte Bedingung erfüllt ist. Der entsprechende Code liegt dabei nicht innerhalb eines weiteren Blocks zwischen geschweiften Klammern, sondern folgt direkt am Ende des `else`-Blocks. Listing 3.28 zeigt den grundlegenden Aufbau von `guard`.

Listing 3.28 Grundlegender Aufbau von `guard`

```
guard <BEDINGUNG> else {
    <AUSZUFÜHRENDER CODE WENN BEDINGUNG NICHT ERFÜLLT IST>
}
<AUSZUFÜHRENDER CODE WENN BEDINGUNG ERFÜLLT IST>
```

Normalerweise würde das dazu führen, dass der Code bei Erfüllung der Bedingung *immer* ausgeführt wird, selbst wenn zuvor der `else`-Block aufgerufen wurde; schließlich folgt ja am Ende von `guard` und so gesehen nach Verlassen des `else`-Blocks trotzdem der zugehörige Code für die Erfüllung der Bedingung. Und dieses Verhalten ist verständlicherweise nicht erwünscht; entweder soll der Code innerhalb des `else`-Blocks ausgeführt werden oder der darauffolgende.

Aus diesem Grund liegt die zweite Besonderheit bei `guard` darin, dass Sie über den `else`-Block die zugrunde liegende Schleife, Abfrage oder Funktion, innerhalb derer sich die `guard`-Abfrage befindet, zwingend verlassen müssen, sodass der darauffolgende Code eben nicht ausgeführt wird. Dazu nutzen Sie entsprechend Control Transfer Statements wie `continue`, `break` oder `return`. Das wiederum bedeutet umgekehrt aber auch, dass Sie eine `guard`-Abfrage nur innerhalb von Schleifen, Abfragen oder Funktionen implementieren können. Wenn Sie beispielsweise innerhalb eines Playgrounds direkt eine `guard`-Abfrage erstellen, kann diese niemals funktionieren, da Sie aus dem `else`-Block heraus nicht verhindern können, dass der nach `guard` folgende Code ausgeführt wird; dazu müsste sich `guard` wie beschrieben in einem separaten Teil Ihres Codes befinden, der verlassen werden kann.

Aufgrund dieser besonderen Funktionsweise von `guard` wird es typischerweise immer dann eingesetzt, wenn man sich entweder sicher ist, dass die gestellte Bedingung in den meisten Fällen erfüllt sein wird, oder dass die zugrunde liegende Funktion nur dann korrekt arbeiten kann, wenn die Bedingung erfüllt ist. In diesen beiden Fällen können Sie mithilfe von `guard` die Bedingung weiterhin prüfen, können die gewünschte Funktionsweise bei Erfüllung der Bedingung aber übersichtlich am Ende des `else`-Blocks aufführen, ohne diese – wie bei `if` – auch in einen Block innerhalb geschweifter Klammern packen zu müssen.

Ein kleines abstraktes Beispiel dazu sehen Sie in Listing 3.29. Die gezeigte Funktionsweise soll den fiktiven Upload dreier Bilder darstellen, die von `1` bis `3` durchnummeriert sind. Dazu steht eine Schleife bereit, die diesen Wertebereich durchläuft und so für jedes Bild einen fiktiven Upload durchführen soll. Allerdings hat diese Funktion keinen Sinn, wenn keine Internetverbindung zur Verfügung steht; diese Verfügbarkeit wird in diesem Beispiel der Einfachheit halber über eine einfache boolesche Variable repräsentiert. Sollte sie `false` sein, soll eine entsprechende Fehlermeldung ausgegeben und die Schleife umgehend verlassen werden, andernfalls kann der Upload erfolgen.

Listing 3.29 Prüfen einer Funktion mit `guard`

```
var internetConnectionAvailable = true
for i in 1...3 {
    guard internetConnectionAvailable else {
        print("Keine Internetverbindung verfügbar.")
        break
    }
    print("Upload von Bild \(i).")
}
// Upload von Bild 1.
// Upload von Bild 2.
// Upload von Bild 3.
```

Würde keine Internetverbindung zur Verfügung stehen, würde der Code nach Ende des `else`-Blocks von `guard` nicht ausgeführt werden, so wie in Listing 3.30 gezeigt.

Listing 3.30 Vorzeitiges Verlassen einer Schleife mittels `guard`

```
var internetConnectionAvailable = false
for i in 1...3 {
    guard internetConnectionAvailable else {
        print("Keine Internetverbindung verfügbar.")
        break
    }
    print("Upload von Bild \(i).")
}
// Keine Internetverbindung verfügbar.
```

3.3 Control Transfer Statements

In Abschnitt 3.2.2, „switch“, wurden bereits zwei erste sogenannte *Control Transfer Statements* vorgestellt: `break` und `fallthrough`. Sie dienen wie alle Control Transfer Statements dazu, die Abfolge von Code und von Befehlen zu beeinflussen. Im Zusammenhang mit Schleifen gibt es ein weiteres Control Transfer Statement namens `continue`, ebenso kann `break` in Schleifen eingesetzt werden. Im Folgenden werde ich Ihnen diese beiden Control Transfer Statements beim Einsatz innerhalb von Schleifen im Detail vorstellen.

3.3.1 Anstoßen eines neuen Schleifendurchlaufs mit continue

Mithilfe des Schlüsselworts `continue` können Sie den Durchlauf einer Schleife umgehend beenden und den nächsten Durchlauf anstoßen. Damit können Sie steuern, ob bestimmte Teile einer Schleife nur unter bestimmten Umständen ausgeführt werden sollen und andernfalls den aktuellen Durchlauf somit umgehend beenden.

Dazu zeigt Listing 3.31 ein Beispiel, in dem für die Zahlen von `1` bis `10` alle ausgegeben werden, die durch `2` teilbar sind. Ist das bei der jeweils aktuellen Zahl des Schleifendurchlaufs nicht der Fall, wird der aktuelle Schleifendurchlauf mithilfe von `continue` beendet und die Schleife mit der nächsten Zahl erneut ausgeführt.

Listing 3.31 Frühzeitiges Verlassen eines Schleifendurchlaufs mittels `continue`

```
for index in 1...10 {
    if index % 2 != 0 {
        continue
    }
    print("\(index) ist durch 2 teilbar.")
}
// 2 ist durch 2 teilbar.
// 4 ist durch 2 teilbar.
// 6 ist durch 2 teilbar.
// 8 ist durch 2 teilbar.
// 10 ist durch 2 teilbar.
```

3.3.2 Verlassen der kompletten Schleife mit break

Wird der Befehl `break` innerhalb einer Schleife aufgerufen, so wird diese umgehend verlassen und kein erneuter Schleifendurchlauf durchgeführt. Damit können Sie unter bestimmten Bedingungen die Ausführung einer Schleife umgehend abbrechen und den nach der Schleife folgenden Code ausführen lassen, ohne darauf zu warten, dass die Schleife bis an ihr Ende durchlaufen wird.

Listing 3.32 zeigt dazu ein kleines Beispiel. Hier wird eine Schleife für die Zahlen 1 bis 10 durchlaufen, gleichzeitig aber außerhalb der Schleife ein Maximalwert von 7 innerhalb der Konstanten `maximumValue` definiert. Wird dieser Maximalwert erreicht, soll die Schleife umgehend verlassen werden.

Listing 3.32 Frühzeitiges Verlassen einer Schleife mittels `break`

```
let maximumValue = 7
for index in 1...10 {
    if index >= maximumValue {
        break
    }
    print("\(index) ist kleiner als \(maximumValue).")
}
print("Schleife verlassen.")
// 1 ist kleiner als 7.
// 2 ist kleiner als 7.
// 3 ist kleiner als 7.
// 4 ist kleiner als 7.
// 5 ist kleiner als 7.
// 6 ist kleiner als 7.
// Schleife verlassen.
```

Weitere Control Transfer Statements

Die bisher vorgestellten Control Transfer Statements `continue`, `break` und `fallthrough` sind nicht die einzigen Befehle, die Ihnen in Swift zur Verfügung stehen. Daneben existieren noch `return` und `throw`. Da diese nur in speziellen Bereichen von Swift zum Einsatz kommen, stelle ich Sie auch erst an passender Stelle im Buch vor.

3.3.3 Labeled Statements

Die in diesem Abschnitt vorgestellten Abfragen und Schleifen lassen sich in Swift auch verschachteln, wie in einigen Listings bereits zu sehen war. Eine Schleife kann somit Abfragen und weitere Schleifen enthalten, genauso wie eine Abfrage selbst auch weitere Abfragen oder Schleifen beinhalten kann.

Diese an sich sehr flexible Möglichkeit kann aber im Zusammenspiel mit Control Transfer Statements wie `continue` oder `break` womöglich zu Problemen führen. Dazu zeigt Listing 3.33 einmal ein passendes Beispiel. Dort sollen die Zahlenwerte von 1 bis 5 mithilfe von `print` ausgegeben werden, allerdings nur dann, wenn eine unabhängig von der Schleife gesetzte Variable namens `printNumericaValue` dem Wert `true` entspricht. Ist das nicht der Fall, soll die Schleife umgehend wieder verlassen werden. Dazu wird `printNumericaValue` mithilfe eines `switch` geprüft; ist es `true`, wird eine zusätzliche Meldung ausgegeben, andernfalls soll die Schleife umgehend mithilfe des Schlüsselworts `break` verlassen werden.

Listing 3.33 Verschachtelte Schleife mit Abfrage

```
var printNumericValue = true
for index in 1...3 {
    switch printNumericValue {
    case true:
        print("Print numerica value.")
    case false:
        break
    }
    print("Value \(index).")
}
// Print numerica value.
// Value 1.
// Print numerica value.
// Value 2.
// Print numerica value.
// Value 3.
```

So weit, so gut. Allerdings wird es nun problematisch, wenn die Variable `printNumericValue` tatsächlich `false` entsprechen sollte. Dann wird zwar der Befehl `break` im entsprechenden Case ausgeführt, doch da dieser sich innerhalb eines `switch` befindet, wird damit nicht die zugrunde liegende Schleife verlassen, sondern lediglich die `switch`-Abfrage; der nachfolgende Code innerhalb der Schleife wird weiterhin ausgeführt (siehe Listing 3.34).

Listing 3.34 Inkorrekte Funktionsweise einer verschachtelten Schleife mit Abfrage

```
var printNumericValue = false
for index in 1...3 {
    switch printNumericValue {
    case true:
        print("Print numerica value.")
    case false:
        break
    }
    print("Value \(index).")
}
// Value 1.
// Value 2.
// Value 3.
```

Auch wenn jetzt die zusätzliche Information `"Print numerica value."` fehlt, wird dennoch die Schleife nicht verlassen, da `break` sich auf den `switch`-Block und nicht auf die Schleife bezieht.

Um solche Probleme zu lösen, stehen in Swift sogenannte *Labeled Statements* zur Verfügung. Dabei wird einer Abfrage oder Schleife ein eigener Bezeichner zugewiesen, über den dann die entsprechende Abfrage oder Schleife innerhalb ihres jeweiligen Code-Blocks direkt mithilfe von Control Transfer Statements angesprochen werden kann.

Um ein Labeled Statement zu erstellen, stellen Sie den Namen des gewünschten Bezeichners der Deklaration der jeweiligen Abfrage oder Schleife voran, gefolgt von einem Doppelpunkt; anschließend wird die Abfrage beziehungsweise Schleife wie gewohnt deklariert. Den grundlegenden Aufbau von Labeled Statements zeigt Listing 3.35.

Listing 3.35 Deklaration eines Labeled Statements

```
<LABELED STATEMENT>: <DEKLARATION DER SCHLEIFE>
```

In dem zuvor gezeigten Beispiel könnte man also der `for-in`-Schleife ein solches Labeled Statement verpassen und dieses anschließend innerhalb der `switch`-Abfrage über den `break`-Befehl ansprechen. Damit würde `break` nicht mehr nur die `switch`-Abfrage, sondern – so wie gewünscht – auch die zugrunde liegende Schleife umgehend verlassen und damit keine einzige Meldung auf der Konsole ausgegeben werden. Den entsprechend angepassten Code zeigt Listing 3.36.

Listing 3.36 Verschachtelte Schleife als Labeled Statement

```
var printNumericValue = false
forLoop: for index in 1...3 {
    switch printNumericValue {
    case true:
        print("Print numerica value.")
    case false:
        break forLoop
    }
    print("Value \(index).")
}
```

Der Schleife wird der Bezeichner `forLoop` zugewiesen, der anschließend dem `break`-Befehl innerhalb der `switch`-Abfrage zugewiesen wird. Damit wird `break` für die `for-in`-Schleife und nicht für die `if`-Abfrage aufgerufen, womit die Schleife im Fall, dass `printNumericValue` dem Wert `false` entspricht, umgehend wieder verlassen wird.

4 Typen in Swift

Typen werden in Swift auf verschiedene Art und Weise deklariert. Es kann sich dabei beispielsweise um sogenannte Structures oder um Klassen handeln (dazu später mehr). Dabei definiert jeder Typ für sich, welche Informationen er enthält und wie man mit ihm arbeiten kann. Einige Typen wie `Int`, `String` und `Bool` haben wir ja bereits in den vorangegangenen Beispielen kennengelernt.

In der Swift Standard Library gibt es eine Vielzahl vorgefertigter Typen zur Programmierung mit Swift. Diese können Sie direkt in Ihrem Code verwenden, ohne dafür irgendetwas tun zu müssen. Zu diesen Typen gehören Typen für Zahlen, Zeichenketten, Wahrheitswerte und viele mehr (eine erste kleine Übersicht über einige der wichtigsten Typen in Swift lieferte Kapitel 2, „Grundlagen der Programmierung“). Diese Typen werden dazu verwendet, Variablen und Konstanten von ihnen zu erstellen und so Werte dieser Typen zu generieren und mit ihnen zu arbeiten. So lassen sich mit einem Zahlentyp Berechnungen durchführen und mithilfe von Zeichenketten Texte ausgeben.

Im Zusammenhang mit der Vorstellung verschiedener Typen werden auch bereits erste wichtige Eigenschaften und Funktionen vorgestellt, die über die entsprechenden Typen aufgerufen und genutzt werden können. Diese werden dabei immer nach einem Punkt am Ende des Variablen- oder Konstantennamens aufgeführt, um anschließend die gewünschte Eigenschaft oder Funktion aufzurufen. Auf Funktionen folgt darüber hinaus ein rundes Klammernpaar, in dem – je nach Funktion – weitere Werte und Parameter übergeben werden. Mehr zu der Funktionsweise von und der Arbeit mit Funktionen erfahren Sie in Kapitel 5, „Funktionen“.

Die folgenden Abschnitte geben Ihnen eine Übersicht über einen großen Teil der in der Swift Standard Library verfügbaren Typen und deren Funktionsweise. Sie werden den gezeigten Typen regelmäßig begegnen, wenn Sie mit Swift programmieren, und Sie erfahren hier alle wichtigen Informationen zu ihnen.

Wie werden Typen definiert?

In diesem Abschnitt geht es, wie beschrieben, um bereits vorhandene Typen, die Sie direkt bei der Programmierung mit Swift nutzen können. Dabei wird ein Typ in der Regel auf vier verschiedene Arten und Weisen definiert:

- In Form einer Enumeration
- In Form einer Structure
- In Form einer Klasse
- In Form eines Protokolls

All diese vier Elemente werden in den kommenden Abschnitten noch im Detail besprochen. Gemein haben sie, dass sie alle einen neuen Typ definieren, den Sie dann als Grundlage für Ihre Variablen und Konstanten verwenden können. Sie dienen somit auch dazu, Ihre eigenen Typen zu erstellen und in Ihren Projekten zu verwenden. Alle Typen aus der Swift Standard Library basieren genauso auf einem dieser Elemente.

Zugriff auf Eigenschaften und Funktionen eines Typs

Jeder Typ verfügt über verschiedene Eigenschaften und Funktionen (von denen im Folgenden sehr viele vorgestellt werden). Um diese Eigenschaften und Funktionen auf Variablen und Konstanten anzuwenden, die einem entsprechenden Typ entsprechen, wird die sogenannte *Punktnotation* verwendet. Das bedeutet, dass nach dem Namen einer Variablen oder Konstanten, auf der eine Eigenschaft oder Funktion des zugehörigen Typs aufgerufen werden soll, ein Punkt gesetzt wird, gefolgt von eben jener Eigenschaft beziehungsweise Funktion (konkrete Beispiele dazu sehen Sie an entsprechenden Stellen in den folgenden Abschnitten). Das lässt sich sogar beliebig verschachteln, man kann auch eine Eigenschaft oder Funktion aufrufen, die auf eine zuvor zugegriffene Eigenschaft oder Funktion folgt. Listing 4.1 zeigt ein paar theoretische Beispiele, um diesen Zugriff auf Eigenschaften und Funktionen zu veranschaulichen.

Listing 4.1 Zugriff auf Eigenschaften und Funktionen von Variablen und Konstanten

```
myVariable.aProperty
myConstant.firstProperty.secondProperty
anotherVariable.aFunction().aProperty
```

4.1 Integer

Integer stellen Zahlen dar, die über keine Nachkommastellen verfügen. Es gibt sie in Swift in vier verschiedenen Größen:

- 8 Bit
- 16 Bit
- 32 Bit
- 64 Bit

Die Größe definiert, welche Werte ein Integer in Swift annehmen kann (dazu gleich mehr). Darüber hinaus unterscheidet Swift zwischen sogenannten *signed* und *unsigned* Integern. Ein Unsigned Integer kann nur null oder einen positiven Wert annehmen, während ein Signed Integer sowohl null als auch einen negativen wie positiven Wert annehmen kann.

Diese beide Faktoren – Größe und signed beziehungsweise unsigned – definieren den Wertebereich, den ein Integer in Swift besitzen kann. Ein Unsigned Integer mit 8 Bit deckt beispielsweise alle Zahlen von 0 bis 255 ab, während ein Signed Integer mit 64 Bit einen Wertebereich von –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 besitzt. Und für all diese Kombinationen existiert in Swift ein passender Integer-Typ; eine Aufstellung dazu gibt Tabelle 4.1.

Tabelle 4.1 Integer-Typen in Swift

Integer-Typ	Größe und Wertebereich
`Int8`	8 Bit Signed Integer, Wertebereich -128 bis 127.
`Int16`	16 Bit Signed Integer, Wertebereich -32.768 bis 32.767.
`Int32`	32 Bit Signed Integer, Wertebereich -2.147.483.648 bis 2.147.483.647.
`Int64`	64 Bit Signed Integer, Wertebereich -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807.
`UInt8`	8 Bit Unsigned Integer, Wertebereich 0 bis 255.
`UInt16`	8 Bit Unsigned Integer, Wertebereich 0 bis 65.535.
`UInt32`	8 Bit Unsigned Integer, Wertebereich 0 bis 4.294.967.295.
`UInt64`	8 Bit Unsigned Integer, Wertebereich 0 bis 18.446.744.073.709.551.615.

Wertebereich von Integern ermitteln

Alle genannten Integer-Typen besitzen zwei Eigenschaften namens `min` und `max`. Wenn Sie diese auf einen der genannten Typen aufrufen, erhalten Sie über `min` den kleinstmöglichen Wert für diesen Typ und über `max` den größtmöglichen (siehe Listing 4.2).

Listing 4.2 Ermitteln der Minimal- und Maximalwerte von Integern

```
print("Minium von Int16: \(Int16.min)")
print("Maximum von UInt32: \(UInt32.max)")
// Minium von Int16: -32768
// Maximum von UInt32: 4294967295
```

Wenn Sie in Swift mit Ganzzahlen arbeiten, können Sie alternativ zu den eben vorgestellten Typen aus Tabelle 4.1 auch einfach einen der zwei folgenden Typen verwenden:

- `Int` (für Signed Integer)
- `UInt` (für Unsigned Integer)

Generell sollten Sie immer die Verwendung von `Int` und `UInt` den zuvor vorgestellten Typen mit expliziter Größe vorziehen und letztere nur dann verwenden, wenn die Größe eine besondere Rolle spielt.

Wann immer Sie in Swift eine Ganzzahl einer Variablen oder Konstanten bei deren Deklaration zuweisen, wird dieser automatisch per Type Inference der Typ `Int` zugewiesen. Wenn Sie stattdessen einen anderen der vorgestellten Integer-Typen verwenden möchten, müssen Sie diesen explizit mittels Type Annotation zuweisen (siehe Listing 4.3).

Listing 4.3 Type Inference und Type Annotation bei Integern

```
let firstInteger = 19
let secondInteger: UInt32 = 99
// firstInteger entspricht Typ Int
// secondInteger entspricht Typ UInt32
```

4.2 Fließkommazahlen

Fließkommazahlen werden in Swift mittels zwei verschiedener Typen abgebildet:

- `Float`
- `Double`

Der Typ `Float` repräsentiert 32-Bit-Fließkommazahlen, während `Double` 64-Bit-Fließkommazahlen darstellen kann. `Double` stellt wenigstens 15 Dezimalstellen dar, während es bei `Float` auch nur sechs sein können. Je nachdem, wie wichtig die Genauigkeit bei einer bestimmten Aufgabe ist, kann man sich davon abhängig für `Double` (sehr wichtig) oder `Float` (nicht so wichtig bis unwichtig) entscheiden. Im Zweifelsfall empfiehlt es sich, `Double` zu verwenden.

Wenn Sie in Swift eine neue Variable oder Konstante erstellen und dieser eine Fließkommazahl zuweisen, dann wird Swift den Typ dieser Variablen beziehungsweise Konstanten per Type Inference automatisch auf `Double` setzen. Möchten Sie stattdessen explizit den Typ `Float` verwenden, so müssen Sie diesen Typ auch explizit mittels Type Annotation zuweisen (siehe Listing 4.4).

Listing 4.4 Type Inference und Type Annotation bei Fließkommazahlen

```
let firstFloatingPointNumber = 19.99
let secondFloatingPointNumber: Float = 99.19
// firstFloatingPointNumber entspricht Typ Double
// secondFloatingPointNumber entspricht Typ Float
```

4.3 Bool

Bei `Bool` handelt es sich um einen sogenannten *Wahrheitswert*. Dieser Typ repräsentiert zwei mögliche Zustände, `true` (wahr) oder `false` (falsch). Das sind auch die einzigen Werte, die eine Variable oder Konstante vom Typ `Bool` annehmen kann.

Wenn Sie einer neu deklarierten Variablen oder Konstanten direkt `true` oder `false` zuweisen, können Sie auf eine explizite Typzuweisung mittels Type Annotation verzichten. Swift erkennt in diesem Fall automatisch, dass es sich bei der neu zu erstellenden Variablen beziehungsweise Konstanten um einen Bool handelt und weist diesen Typ mittels Type Inference zu.

4.4 String

Mithilfe von Strings bilden Sie sogenannte Zeichenketten ab. Jegliche Form von Text wird in Swift mithilfe des zugehörigen Typs `String` abgebildet, der darüber hinaus eine Vielzahl an Funktionen mitbringt, um mit Strings zu arbeiten und diese zu manipulieren, zu verändern und auszuwerten.

4.4.1 Erstellen eines Strings

Einen neuen String in Swift erstellen Sie, indem Sie eine neue Variable oder Konstante deklarieren und dieser dann eine gewünschte Zeichenkette zuweisen. Diese wird dabei von doppelten Anführungszeichen umfasst (siehe Listing 4.5).

Listing 4.5 Erstellen eines neuen Strings

```
var aString = "Ein neuer String"
```

Die Variable `aString` entspricht hier somit der Zeichenkette `"Ein neuer String"`.

Möchten Sie einen neuen leeren String erstellen, so gibt es dafür zwei Möglichkeiten: Entweder weisen Sie der zugehörigen Variablen oder Konstanten eine leere Zeichenkette zu (indem Sie direkt hintereinander die öffnenden und schließenden Anführungszeichen ohne Inhalt dazwischen setzen) oder indem Sie die sogenannte Initializer Syntax von Swift verwenden. Diese steht in Swift bei allen Typen zur Verfügung, mehr dazu erfahren Sie in Kapitel 8, „Initialisierung“. Listing 4.6 zeigt beide Wege zum Erstellen eines neuen leeren Strings, zunächst mittels Zuweisung einer leeren Zeichenkette, dann mittels der Initializer Syntax.

Listing 4.6 Erstellen eines neuen leeren Strings

```
var anotherString = ""
var initializedString = String()
```

4.4.2 Zusammenfügen von Strings

Strings können in Swift manipuliert werden, sofern sie einer Variablen und nicht einer Konstanten zugewiesen sind. Sie können den Berechnungs- und Zuweisungsoperator += verwenden, um einen String um einen zusätzlichen String zu ergänzen, oder mithilfe des Berechnungsoperators + mehrere Strings kombinieren, um daraus einen neuen String zu erstellen. In Listing 4.7 sehen Sie ein Beispiel dazu.

Listing 4.7 Verändern eines Strings

```
let myFirstName = "Thomas"
let myLastName = "Sillmann"
let myName = myFirstName + " " + myLastName
print("myName entspricht \(myName)")
var greeting = "Mein Name ist "
greeting += myName
print("\(greeting)")
// myName entspricht Thomas Sillmann
// Mein Name ist Thomas Sillmann
```

Da es sich bei `greeting` um eine Variable handelt, kann der ihr bereits zugewiesene String auch im Nachhinein noch manipuliert und verändert werden; bei Konstanten

ist das nicht möglich. So würde ein ergänzender Versuch, den Namen der Konstanten `myName` auf die gleiche Art und Weise zu ändern, zu einem Compiler-Fehler führen. In Listing 4.8 wird dieses Verhalten demonstriert, indem versucht wird, `myName` am Ende um einen Punkt zu ergänzen, was fehlschlägt.

Listing 4.8 Das Ändern von Konstanten ist nicht möglich.

```
myName += "."
// Compiler-Fehler: Konstanten können nicht verändert werden.
```

Man spricht hierbei auch von *String Mutability* und *String Immutability*. Variablen vom Typ `String` können jederzeit verändert werden (String Mutability), während ein einmal zugewiesener String zu einer Konstanten niemals wieder angepasst werden kann (String Immutability).

Daneben verfügt der Typ `String` über eine Funktion namens `append(_:)` (mehr zu Funktionen erfahren Sie in Kapitel 5, „Funktionen“). Mithilfe dieser Funktion können Sie einem String einen `Character` zuweisen, der dann an das Ende des Strings angefügt wird (siehe Listing 4.9).

Listing 4.9 Ergänzen eines Strings um einen Character

```
var hello = "Hallo"
hello.append("!")
print("\(hello)")
// "Hallo!"
```

Wichtig ist dabei, zu beachten, dass diese Funktion nur dann auf einem String aufgerufen werden kann, wenn dieser als Variable deklariert ist; Konstanten können diese Funktion aufgrund der String Immutability nicht nutzen.

Character

`Character` ist ein Typ der Swift Standard Library, genau wie `String` auch. Statt komplexer und langer Zeichenketten verweist ein Character aber lediglich auf exakt ein Zeichen (wie im eben gezeigten Beispiel das Ausrufezeichen). Somit setzt sich ein String wiederum schlicht aus mehreren Charactern zusammen.

Wenn Sie in Swift explizit eine Variable oder Konstante vom Typ `Character` erstellen möchten, reicht es nicht aus, einfach einer neuen Variablen beziehungsweise Konstanten eine Zeichenkette mit exakt einem Zeichen zuzuweisen; anhand der Type Inference wird Swift dennoch annehmen, dass es sich bei dem neu erstellten Wert nichtsdestoweniger um einen vollwertigen String handelt. In diesen Fällen müssen Sie den gewünschten Typ also explizit mittels Type Annotation angeben, so wie in Listing 4.10 zu sehen.

Listing 4.10 Erstellen eines Characters

```
let firstCharacter = "C"
let secondCharacter: Character = "h"
```

Bei der ersten Konstanten `firstCharacter` handelt es sich um einen `String`. Nur die zweite, der explizit der Typ `Character` zugewiesen wird, ist auch tatsächlich vom Typ `Character`.

Wie beschrieben, verweist ein `Character` auf exakt ein Zeichen. Sollten Sie versuchen, einem Character mehr als ein Zeichen zuzuweisen, endet das in einem Compiler-Fehler.

4.4.3 Character auslesen

Mithilfe einer `for-in`-Schleife können Sie die einzelnen Character, aus denen sich ein String zusammensetzt, nacheinander auslesen. Dazu übergeben sie den gewünschten String als Wertebereich für die `for-in`-Schleife. Damit wird die Schleife für jeden einzelnen Character des Strings durchlaufen und jeder Character dem von Ihnen definierten Platzhalter übergeben. In Listing 4.11 sehen Sie ein Beispiel dazu.

Listing 4.11 Character auslesen

```
let myName = "Thomas"
for character in myName {
    print("\(character)")
}
// T
// h
// o
// m
// a
// s
```

4.4.4 Character mittels Index auslesen

Der Typ `String` verfügt über verschiedene Funktionen, um auf bestimmte Character innerhalb eines Strings zugreifen zu können. Dabei bedienen sich diese Funktionen eines sogenannten *Index*. Dieser Index beginnt bei `0` und verweist damit auf den ersten Character eines Strings, Index `1` verweist dann auf den zweiten Character, Index `2` auf den dritten und so weiter.

Die Eigenschaft `startIndex` von `String` liefert den Index für den ersten Character des Strings zurück. Die Eigenschaft `endIndex` liefert den Index *nach* dem letzten Character des zugehörigen Strings. Dieses kleine Detail, dass `endIndex` sich auf den Index *nach*

dem letzten Character bezieht, ist auch immens wichtig, da sich an der Position des `endIndex` des zugehörigen Strings kein Character mehr befindet; ein Versuch, auf diesen nicht vorhandenen Index zuzugreifen, würde umgehend zum Absturz Ihrer Anwendung führen. Der Index des letzten Characters eines Strings entspräche somit vielmehr `endIndex` minus eins. Einzige Ausnahme: Handelt es sich um einen leeren String, dann sind `startIndex` und `endIndex` identisch.

Daneben verfügt der Typ `String` noch über zusätzliche Funktionen zum Ermitteln weiterer Indexe eines Strings. Zunächst einmal sind da `index(before:)` und `index(after:)`. Diese liefern je den Index *vor* dem übergebenen Index beziehungsweise *nach* dem übergebenen Index. Letztere Funktion kann somit beispielsweise ideal im Zusammenspiel mit der Eigenschaft `endIndex` verwendet werden, um den korrekten Index des letzten Characters eines Strings zu erhalten (da `endIndex` wie beschrieben auf den Index *nach* dem letzten Character eines Strings verweist).

Noch flexibler gestaltet sich die Funktion `index(_:offsetBy:)`. Über diese wird zunächst ein Startindex für den jeweiligen String angegeben, gefolgt von einem Offset in Form eines Integers. Die Funktion liefert anschließend den Index des Characters zurück, der sich aus der Addition von Startindex und Offset ergibt.

In Listing 4.12 sehen Sie einige Beispiele zur Anwendung der genannten Eigenschaften und Funktionen, um so verschiedene Indexe von Charactern eines Strings zu erhalten.

Listing 4.12 Indexe zu Charactern eines Strings ermitteln

```
let helloWorld = "Hallo Welt!"
let startIndex = helloWorld.startIndex
let endIndex = helloWorld.endIndex
let indexBeforeEndIndex = helloWorld.index(before: endIndex)
let indexAfterStartIndex = helloWorld.index(after: startIndex)
let indexWithOffset = helloWorld.index(startIndex, offsetBy: 4)
```

Diese so generierten Indexe können nun dazu verwendet werden, die zugehörigen Character eines Strings auszulesen. Dazu verfügt der Typ `String` über ein sogenanntes *Subscript*. Dabei handelt es sich um Funktionen, die aufgerufen werden, indem man innerhalb von eckigen Klammern am Ende einer Variablen oder Konstanten bestimmte Werte übergibt, die sodann einen Befehl ausführen. Bei `String` können Sie über diese Subscript-Syntax einen der zuvor erstellten Indexe übergeben, und Sie erhalten dafür den zugehörigen Character zum passenden Index zurück. In Listing 4.13 sehen Sie einige beispielhafte Anwendungen eines solchen Subscripts mit den zuvor in Listing 4.12 erstellten Indexen.

Listing 4.13 Auslesen von Charactern eines Strings mithilfe eines Subscripts

```
let startCharacter = helloWorld[startIndex]
let endCharacter = helloWorld[indexBeforeEndIndex]
let characterAfterStartIndex = helloWorld[indexAfterStartIndex]
let characterWithOffset = helloWorld[indexWithOffset]
```

```
print("\(startCharacter)")
print("\(endCharacter)")
print("\(characterAfterStartIndex)")
print("\(characterWithOffset)")
// H
// !
// a
// o
```

Mehr zum Thema Subscripts erfahren Sie in Kapitel 7, „Eigenschaften und Funktionen von Typen“.

Der Typ Index

Bei dem in diesem Abschnitt vorgestellten Index handelt es sich nicht um einen einfachen Integer, sondern um einen ganz eigenen Typ namens `Index`, der im Typ `String` definiert ist. Dieser ist ein wenig komplexer und mächtiger als ein einfacher Integer und verfügt über zusätzliche Funktionen. Daher ist es nicht möglich, im gezeigten Subscript einfach einen Integer wie `0` oder `4` zu übergeben, da das direkt zu einem Compiler-Fehler führen würde; in diesem Subscript sind nur Werte vom genannten Typ `Index` erlaubt. Daher benötigen Sie auch die gezeigten Eigenschaften und Methoden wie `startIndex`, `index(after:)` oder `index(_:offsetBy:)`, um darüber passende Index-Objekte zu erstellen, die Sie dann innerhalb des Subscripts verwenden können.

4.4.5 Character entfernen und hinzufügen

Mithilfe des im vorigen Abschnitt vorgestellten Typs `Index` ist es ebenfalls möglich, Character aus Strings zu entfernen beziehungsweise neue Character hinzuzufügen. Um diese Aufgaben zu erfüllen, stellt der Typ `String` einige passende Funktionen bereit.

Hinzufügen von Charactern

Für das Hinzufügen von Charactern gibt es die Funktionen `insert(_:at:)` und `insert(contentsOf:at:)`. Mit ersterer wird ein einzelner Character einem String an einer gewünschten Indexposition hinzugefügt, mit letzter mehrere Character auf einmal. Als Erstes werden dabei in beiden Fällen der beziehungsweise die gewünschten Character aufgeführt, die einem String hinzugefügt werden sollen, während als Zweites der Index innerhalb des Strings anzugeben ist, an dem die zuvor genannten Character eingefügt werden sollen. Listing 4.14 zeigt zwei Beispiele zur Verwendung beider Methoden.

Listing 4.14 Hinzufügen von Charactern zu einem String

```
var greeting = "Hallo"
greeting.insert("!", at: greeting.endIndex)
print("1. Änderung: \(greeting)")
// Hallo!

let greetingUpdateText = " Welt"
let greetingIndexBeforeEndIndex = greeting.index(before: greeting.endIndex)
greeting.insert(contentsOf: greetingUpdateText, at: greetingIndexBeforeEndIndex)
print("2. Änderung: \(greeting)")
// Hallo Welt!
```

Im ersten Schritt wird an das Ende der Variablen `greeting` ein einzelner Character in Form eines Ausrufezeichens angefügt. Im zweiten Schritt soll dann ein neuer String, der in der Konstanten `greetingUpdateText` gespeichert ist, mithilfe der Methode `insert(contentsOf:at:)` der Variablen `greeting` hinzugefügt werden. Dazu wird mithilfe der Methode `index(before:)` die Position innerhalb des Strings vor dem letzten Zeichen (sprich dem Ausrufezeichen nach "Hallo") ermittelt und in der Konstanten `greetingIndexBeforeEndIndex` gespeichert. Dieser Index wird dann zusammen mit dem neu zu ergänzenden Text bei Aufruf der Methode `insert(contentsOf:at:)` genutzt, um den Text ans Ende des Strings vor dem Ausrufezeichen einzufügen.

Entfernen von Charactern

Auch für das Entfernen von Charactern aus einem String stehen zwei Funktionen zur Verfügung: Mithilfe von `remove(at:)` entfernen Sie genau einen Character an einer gewünschten Indexposition, während Sie mit `removeSubrange(_:)` eine Range an Indexen übergeben, die aus dem gewünschten String entfernt werden sollen. Eine beispielhafte Umsetzung beider Methoden sehen Sie in Listing 4.15. Dabei wird an die zuvor erstellte `greeting`-Variable angeknüpft und diese entsprechend verändert und angepasst.

Listing 4.15 Entfernen von Charactern aus einem String

```
greeting.remove(at: greeting.startIndex)
print("1. Änderung: \(greeting)")
let rangeToRemove = greeting.startIndex...greeting.index(greeting.startIndex,
offsetBy: 4)
greeting.removeSubrange(rangeToRemove)
print("2. Änderung: \(greeting)")
// 1. Änderung: allo Welt!
// 2. Änderung: Welt!
```

Wichtig ist bei der Funktion `removeSubrange(_:)`, wie beschrieben, dass die Range keine Integer, sondern Indexe vom Typ `Index` darstellt. Die Range wird in der Konstanten `rangeToRemove` gespeichert und beginnt mit dem Startindex von `greeting` und endet mit einschließlich der vierten Indexstelle.

4.4.6 Anzahl der Character zählen

Eine weitere Eigenschaft in Bezug auf die Character eines Strings lautet `count`. Diese kann direkt auf den gewünschten String angewendet werden. Darüber erhalten Sie die Anzahl der Character dieses Strings als Integer zurück (siehe Listing 4.16).

Listing 4.16 Anzahl Character eines Strings zählen

```
let myName = "Thomas Sillmann"
let myNameCharacterNumber = myName.count
print("Anzahl Character in myName: \(myNameCharacterNumber)")
// Anzahl Character in myName: 15
```

4.4.7 Präfix und Suffix prüfen

Der Typ `String` bringt zwei Funktionen mit, mit deren Hilfe überprüft werden kann, ob ein String über ein bestimmtes Präfix beziehungsweise Suffix verfügt. Diese Funktionen nennen sich `hasPrefix(_:)` beziehungsweise `hasSuffix(_:)`. Sie werden auf dem String aufgerufen, für den geprüft werden soll, ob ein entsprechendes Präfix oder Suffix existiert. Die Funktionen erhalten das zu vergleichende Präfix beziehungsweise Suffix als Parameter. Listing 4.17 zeigt ein Beispiel dazu.

Listing 4.17 Prüfen auf Präfix und Suffix

```
let prefix = "Präfix"
if prefix.hasPrefix("Prä") {
    print("\(prefix) beginnt mit Prä.")
}
let suffix = "Suffix"
if suffix.hasSuffix("fix") {
    print("\(suffix) endet mit fix.")
}
// Präfix beginnt mit Prä.
// Suffix endet mit fix.
```

Die Funktionen `hasPrefix(_:)` und `hasSuffix(_:)` liefern einen booleschen Wert zurück, der entweder `true` ist, sollte das jeweilige Präfix oder Suffix im abgefragten String existieren, oder `false`, falls dem nicht so ist.

4.4.8 String Interpolation

Eine mächtige Funktion von Strings haben wir bereits an mehreren Stellen in diesem Buch kennengelernt, die sogenannte *String Interpolation*. Diese erlaubt es, den Wert von Variablen und Konstanten in eine Zeichenkette einzubauen. Dazu muss innerhalb des Strings ein Platzhalter für die Variable beziehungsweise Konstante mithilfe

von \() erzeugt werden, innerhalb der runden Klammern notiert man dann den Namen der Variablen beziehungsweise Konstanten. Ein einfaches Beispiel dazu sehen Sie in Listing 4.18.

Listing 4.18 Einsatz von String Interpolation

```
let myFirstName = "Thomas"
let myLastName = "Sillmann"
let myFullName = "\(myFirstName) \(myLastName)"
print("\(myFullName)")
Thomas Sillmann
```

Hier wird String Interpolation direkt an zwei Stellen eingesetzt. Zunächst einmal, um die Konstante `myFullName` aus den Konstanten `myFirstName` und `myLastName` zu erstellen, und anschließend, um `myFullName` mithilfe des `print()`-Befehls auszugeben.

String Interpolation kann dazu genutzt werden, Strings aus den Werten anderer Variablen und Konstanten zusammenzustellen. Doch darüber hinaus können Sie noch weitere Operationen mithilfe von String Interpolation durchführen, beispielsweise Berechnungen (siehe Listing 4.19).

Listing 4.19 Berechnung mithilfe von String Interpolation

```
let calculation = "19 * 99 = \(19 * 99)"
print("\(calculation)")
// 19 * 99 = 1881
```

4.5 Array

Mithilfe von Arrays speichern Sie mehrere verschiedene Werte ein und desselben Typs. Ein Array kann also beispielsweise mehrere verschiedene Strings oder mehrere verschiedene Integer enthalten. Auf die einzelnen Werte eines Arrays können Sie mithilfe eines Index zugreifen, der bei `0` für das erste Element beginnt und für jedes weitere Element um eins hochgezählt wird.

Arrays in Swift sind vom Typ `Array<Element>`, wobei `Element` für den Typ steht, deren Werte das jeweilige Array enthalten kann. Statt dieser doch recht umfangreichen Typbezeichnung gibt es in Swift aber auch eine Kurzschreibweise für Arrays, die als *Array Type Shorthand Syntax* bezeichnet wird. Durch diese setzt sich der Name des Array-Typs aus eckigen Klammern zusammen, zwischen denen der Typ definiert wird, mit dessen Werten das Array umgehen kann, zum Beispiel `[Element]`. Da diese Kurzschreibweise für Arrays die bevorzugte in Swift ist, werde ich sie in diesem Buch ausschließlich verwenden. Technisch gesehen sind aber beide Varianten – `Array<Element>` und `[Element]` – vollkommen identisch.

Typsicherheit von Arrays

Im Gegensatz zu vielen anderen Programmiersprachen sind Arrays in Swift grundsätzlich *typsicher*. Das bedeutet, dass ein Array nur Objekte eines festgelegten Typs enthalten kann, also beispielsweise entweder nur String, nur Integer oder nur Double, aber keine Mischung aus diesen. So kann ein Array mit Strings keine Integer enthalten und diese können auch nicht später hinzugefügt werden.

Diese Typsicherheit wird bereits vom Compiler abgefangen, weshalb das eben beschriebene Hinzufügen eines Integers zu einem String-Array in einem Compiler-Fehler endet.

Zwar gibt es in Swift durchaus Möglichkeiten, diese strikte Typsicherheit zu umgehen (die werden wir auch noch im Laufe des Buches kennenlernen), ganz generell gilt aber das beschriebene Verhalten.

Wie bei allen anderen Typen in Swift ist auch bei Arrays zu beachten, dass diese entweder *mutable* (veränderbar) oder *immutable* (unveränderlich) sein können. In ersterem Fall kann ein bereits erstelltes Array im Nachhinein noch verändert werden (beispielsweise durch das Entfernen bestehender oder Hinzufügen neuer Elemente), in letzterem Fall nicht. Dabei werden mutable Arrays immer als Variablen mithilfe des Schlüsselworts `var` erstellt, während immutable Arrays mittels Konstanten und dem Schlüsselwort `let` umgesetzt werden.

4.5.1 Erstellen eines Arrays

Es gibt in Swift verschiedene Wege, ein neues Array zu erstellen. Am einfachsten ist es, einer neuen Variablen oder Konstanten direkt ein Array zuzuweisen. Dabei beginnt die Erstellung des zuzuweisenden Arrays mit einer geöffneten eckigen Klammer, gefolgt von den Werten, die das Array enthalten soll (die jeweiligen Werte werden dabei durch ein Komma getrennt). Nach dem letzten Wert folgt dann eine schließende eckige Klammer. Ein Beispiel dazu zeigt Listing 4.20.

Listing 4.20 Erstellen eines Arrays mit einem Standardwert

```
let stringArray = ["Eins", "Zwei", "Drei"]
```

Da es sich bei den Werten innerhalb des Arrays um Strings handelt, handelt es sich bei dem Array auch um ein String-Array vom Typ `[String]`. Das müssen Sie an dieser Stelle nicht explizit mittels Type Annotation angeben, da Swift diese Tatsache selbst ableiten kann, da die Elemente innerhalb des Arrays allesamt vom Typ `String` sind. Dieses Array kann somit ausschließlich mit Strings umgehen, aber beispielsweise nicht mit Ganz- oder Fließkommazahlen.

Um stattdessen ein neues leeres Array zu erstellen, gibt es in Swift zwei Möglichkeiten. Die eine ist die Verwendung der sogenannten Initializer Syntax (mehr zur Initialisierung erfahren Sie in Kapitel 8, „Initialisierung“). Dabei weisen Sie einer Variablen oder Konstanten den gewünschten Array-Typ zu (beispielsweise `[Int]` für ein Integer-Array), gefolgt von einer geöffneten und geschlossenen runden Klammer. Alternativ dazu können Sie einer Variablen oder Konstanten einfach ein leeres Array ohne jegliche Werte zuweisen, müssen dann aber zwingend per Type Annotation den Array-Typ deklarieren. Da schließlich das zugewiesene Array über keinerlei Werte verfügt, kann Swift ansonsten auch nicht nachvollziehen, welchen Typ es per Type Inference für die entsprechende Variable oder Konstante deklarieren soll.

Beide genannten Vorgehensweisen zum Erstellen eines neuen leeren Arrays zeigt einmal Listing 4.21.

Listing 4.21 Erstellen eines neuen leeren Arrays

```
// Initializer Syntax:
let initializedIntArray = [Int]()
// Zuweisen eines leeren Arrays mit Type Annotation
let emptyIntArray: [Int] = []
```

Alternativ zu der eben gezeigten Initializer Syntax bringt der Typ Array auch noch eine weitere Funktion mit, um ein neues Array zu erstellen. Dabei geben Sie die Anzahl der Elemente an, über die das Array verfügen soll, sowie einen Standardwert. Anschließend wird ein neues Array erstellt, das die genannte Anzahl an Werten besitzt, wobei bei jedem Wert der genannte Standardwert verwendet wird. Listing 4.22 zeigt die Verwendung dieser Funktion, indem ein Double-Array mit fünf Elementen erstellt wird, die alle den Wert `19.99` enthalten.

Listing 4.22 Erstellen eines Arrays mit Standardwerten

```
let doubleArray = Array(repeating: 19.99, count: 5)
```

4.5.2 Zusammenfügen von Arrays

Mithilfe des Zuweisungs- und Berechnungsoperators += beziehungsweise des Berechnungsoperators + ist es möglich, Arrays miteinander zu verbinden und zusammenzufügen. Mithilfe des Operators += wird dabei dem Array auf der linken Seite des Operators der Inhalt des Arrays auf der rechten Seite hinzugefügt, während mit dem +-Operator mehrere Arrays zu einem neuen verbunden werden können. Die praktische Anwendung beider Operatoren zeigt Listing 4.23.

Listing 4.23 Zusammenfügen von Arrays

```
var firstIntArray = [18, 19, 20]
let secondIntArray = [98, 99, 100]
```

```
let intArray = firstIntArray + secondIntArray
print("intArray: \(intArray)")
firstIntArray += secondIntArray
print("firstIntArray: \(firstIntArray)")
// intArray: [18, 19, 20, 98, 99, 100]
// firstIntArray: [18, 19, 20, 98, 99, 100]
```

4.5.3 Inhalte eines Arrays leeren

Um alle Inhalte und Werte aus einem bestehenden Array zu entfernen, reicht es aus, dem Array ein neues leeres Array zuzuweisen, so wie in Listing 4.24 zu sehen. Dabei ist lediglich wichtig, zu beachten, dass dieses Vorgehen ausschließlich bei Arrays funktioniert, die als Variable deklariert sind und somit *mutable* (also veränderbar) sind; bei Konstanten funktioniert das gezeigte Vorgehen nicht.

Listing 4.24 Leeren eines bestehenden Arrays

```
var existingArray = [1, 2, 3]
existingArray = []
```

Daneben steht auch die Funktion `removeAll()` zur Verfügung, die auf einem mutable Array aufgerufen werden kann, um daraus alle Elemente zu entfernen (siehe Listing 4.25; das Ergebnis ist das gleiche wie in Listing 4.24).

Listing 4.25 Leeren eines bestehenden Arrays mittels `removeAll()`

```
existingArray.removeAll()
```

4.5.4 Prüfen, ob ein Array leer ist

Der Typ `Array` verfügt über eine Eigenschaft namens `isEmpty`. Diese liefert einen booleschen Wert zurück, der darüber informiert, ob das Array einen Inhalt besitzt oder nicht. Man erhält `true`, sollte das abgefragte Array leer sein, andernfalls `false`. Listing 4.26 zeigt eine beispielhafte Anwendung dieser Funktion.

Listing 4.26 Prüfen, ob ein Array leer ist

```
let emptyArray = [String]()
let notEmptyArray = ["Not", "Empty"]
if emptyArray.isEmpty {
    print("emptyArray ist leer.")
}
if notEmptyArray.isEmpty {
    print("notEmptyArray ist leer.")
}
// emptyArray ist leer.
```

4.5.5 Anzahl der Elemente eines Arrays zählen

Der Typ `Array` verfügt über eine Eigenschaft namens `count`, über die man die Anzahl der Werte des Arrays, über das man diese Eigenschaft aufruft, in Form eines Integers erhält. Somit lässt sich mit `count` die aktuelle Größe eines Arrays ermitteln. Listing 4.27 zeigt die beispielhafte Anwendung dieser Funktion.

Listing 4.27 Zählen der Elemente eines Arrays

```
let names = ["Thomas", "Michaela", "Tobias"]
print("names enthält \(names.count) Werte.")
// names enthält 3 Werte.
```

4.5.6 Zugriff auf die Elemente eines Arrays

Um auf die verschiedenen Elemente eines Arrays zuzugreifen, wird ein sogenannter Index genutzt. Dieser beginnt bei `0` für das erste Element, `1` verweist auf das zweite, `2` auf das dritte und so weiter. Um auf das Element des gewünschten Index eines Arrays zuzugreifen, wird zunächst der Name des Arrays geschrieben, gefolgt von eckigen Klammern. Innerhalb der eckigen Klammern folgt die Angabe des Index, dessen Objekt ausgelesen werden soll. Es handelt sich dabei um eine sogenannte Subscript-Funktion, in der im Falle von Arrays die Werte innerhalb des Arrays ausgelesen werden können (mehr zum Thema Subscript erfahren Sie in Kapitel 7, „Eigenschaften und Funktionen von Typen“). Listing 4.28 zeigt ein Beispiel dazu.

Listing 4.28 Auslesen der Werte eines Arrays

```
let names = ["Thomas", "Michaela", "Tobias"]
let thomas = names[0]
let michaela = names[1]
let tobias = names[2]
print("\(thomas)")
print("\(michaela)")
print("\(tobias)")
// Thomas
// Michaela
// Tobias
```

Wichtig ist dabei, auf einen Index zuzugreifen, der auch tatsächlich im Array existiert. Würde man im Falle des `names`-Arrays aus dem Listing von eben versuchen, einen Index von `3` oder höher auszulesen, würde das zu einem Absturz des Programms führen, da ein solcher Index in diesem Array nicht existiert. Daher kann es sinnvoll sein, vor dem Auslesen von Werten aus einem Array die Anzahl der Elemente dieses Arrays zu ermitteln, um sicherzustellen, dass der gewünschte Index auch tatsächlich existiert und entsprechend abgefragt werden kann.

4.5.7 Neue Elemente zu einem Array hinzufügen

Ist ein Array als Variable deklariert (und damit mutable), können diesem neue Elemente hinzugefügt werden. Für diesen Zweck stehen drei mögliche Wege zur Verfügung.

Der erste besteht in der Verwendung der `Array`-Funktion `append(_:)`. Diese rufen Sie auf dem zu erweiternden Array auf und übergeben anschließend innerhalb der runden Klammern den neuen Wert für dieses Array. Achten Sie dabei darauf, dass dieser Wert dem Typ des Arrays entspricht, also dass es sich beispielsweise im Falle eines String-Arrays auch um einen String handelt. Der übergebene Wert wird dann als neues Element an das Ende des Arrays angefügt. Listing 4.29 zeigt ein Beispiel dazu.

Listing 4.29 Hinzufügen eines Elements zu einem Array mittels `append(_:)`

```
var names = ["Thomas", "Michaela", "Tobias"]
print("Namen 1: \(names)")
names.append("Luisa")
print("Namen 2: \(names)")
// Namen 1: ["Thomas", "Michaela", "Tobias"]
// Namen 2: ["Thomas", "Michaela", "Tobias", "Luisa"]
```

Eine alternative Möglichkeit zur Funktion `append(_:)` stellt der Zuweisungs- und Berechnungsoperator `+=` dar. Damit können Sie ein bestehendes Array um die Werte eines anderen Arrays erweitern, die dann ebenfalls an das Ende des bestehenden Arrays angefügt werden. Mehr dazu erfahren Sie in Abschnitt 4.5.2, „Zusammenfügen von Arrays“.

Die letzte und zugleich flexibelste Methode zum Hinzufügen neuer Werte zu einem Array besteht in einer weiteren `Array`-Funktion namens `insert(_:at:)`. Dieser übergeben Sie einerseits den Wert, den Sie dem entsprechenden Array hinzufügen möchten, und andererseits den Index, an dessen Stelle der entsprechende Wert innerhalb des Arrays eingefügt werden soll. Dabei verschiebt sich der Wert, der sich zuvor an der entsprechenden Indexstelle innerhalb des Arrays befand, um eins nach oben, das Gleiche gilt auch für alle nachfolgenden Werte des Arrays.

Auf diese Art und Weise wird beispielsweise in Listing 4.30 ein neuer Name zu Beginn des bereits vorgestellten `names`-Arrays angefügt.

Listing 4.30 Hinzufügen eines Elements zu einem Array mittels `insert(_:at:)`

```
names.insert("Fine", at: 0)
print("Namen 3: \(names)")
// Namen 3: ["Fine", "Thomas", "Michaela", "Tobias", "Luisa"]
```

4.5.8 Bestehende Elemente aus einem Array entfernen

Analog zum Hinzufügen neuer Elemente zu einem bestehenden Array lassen sich auch bereits vorhandene Elemente eines Arrays wieder entfernen. Auch hierbei gibt es in Swift verschiedene Möglichkeiten, dieses Vorhaben umzusetzen.

Um beispielsweise das letzte Element eines Arrays zu entfernen, kann die Funktion `removeLast()` verwendet werden, die auf einem Array aufgerufen werden kann. Wie der Name bereits andeutet, wird damit das aktuell letzte Element des jeweiligen Arrays entfernt. Bei Verwendung der Funktion ist lediglich darauf zu achten, dass mindestens ein Element innerhalb des Arrays existiert. Wenn das Array leer ist und die Funktion `removeLast()` wird dennoch aufgerufen, führt das zum Absturz der Anwendung.

Als Gegenstück zur Funktion `removeLast()` gibt es die Funktion `removeFirst()`, mit der das erste Element eines Arrays entfernt werden kann. Auch hier ist darauf zu achten, dass das Array, auf dem diese Funktion aufgerufen wird, wenigstens über ein Element verfügt, das entfernt werden kann.

Eine weitere Funktion des Typs `Array` zum Entfernen bereits bestehender Elemente ist `remove(at:)`. Dabei übergeben Sie den Index des gewünschten zu entfernenden Elements als Parameter. Auch hierbei ist zu beachten, dass der angegebene Index in dem zugehörigen Array vorhanden sein muss. Der Versuch, ein Element eines Index zu entfernen, den es in dem Array gar nicht gibt, führt zum direkten Absturz des Programms.

Listing 4.31 zeigt einmal beispielhaft den Einsatz der vorgestellten Funktionen `removeLast()` und `remove(at:)`.

Listing 4.31 Entfernen von bestehenden Elementen aus einem Array mittels `removeLast()` und `remove(at:)`

```
var names = ["Fine", "Thomas", "Michaela", "Tobias", "Luisa"]
print("Namen 1: \(names)")
names.removeLast()
print("Namen 2: \(names)")
names.remove(at: 1)
print("Namen 3: \(names)")
// Namen 1: ["Fine", "Thomas", "Michaela", "Tobias", "Luisa"]
// Namen 2: ["Fine", "Thomas", "Michaela", "Tobias"]
// Namen 3: ["Fine", "Michaela", "Tobias"]
```

Eine weitere mögliche Methode zum Entfernen von Elementen aus einem Array ist `removeAll()`. Damit werden – wie der Name bereits andeutet – alle Elemente aus dem Array entfernt. Sie können diese Methode auch gefahrlos aufrufen, sollte das betreffende Array keinerlei Elemente besitzen; in diesem Fall geschieht schlicht gar nichts.

Bei all den gezeigten Funktionen und Möglichkeiten zum Entfernen vorhandener Elemente aus einem Array muss allerdings immer bedacht werden, dass diese nur ange-

wendet werden können, wenn das betreffende Array in einer Variablen gespeichert und somit mutable (also veränderbar) ist. Auf ein Array, das als Konstante deklariert ist, können all die gezeigten Funktionen nicht angewendet werden, ohne dass es zu einem Compiler-Fehler kommt.

4.5.9 Bestehende Elemente eines Arrays ersetzen

Neben dem Hinzufügen neuer und dem Entfernen bestehender Elemente gibt es auch die Möglichkeit, vorhandene Elemente eines Arrays mit neuen Werten zu überschreiben. Dabei gilt ebenfalls, dass dieses Verhalten nur dann eingesetzt werden kann, wenn das entsprechende Array als Variable deklariert ist, andernfalls ist das Array unveränderlich.

Um ein vorhandenes Element eines mutable Arrays durch ein neues zu ersetzen, greifen Sie per Subscript auf das gewünschte Element des Arrays zu und ersetzen es anschließend mithilfe des Zuweisungsoperators = durch den gewünschten neuen Wert. In Listing 4.32 sehen Sie ein Beispiel dazu.

Listing 4.32 Ersetzen bestehender Elemente eines Arrays

```
names = ["Thomas", "Michaela", "Tobias"]
names[2] = "Luisa"
print("Namen: \(names)")
// Namen: ["Thomas", "Michaela", "Luisa"]
```

Auf diese Art und Weise können Sie sogar mehrere Elemente auf einmal ersetzen. Dazu geben Sie für den Index innerhalb des Subscripts einen Index-Wertebereich in Form einer Range an und weisen diesem anschließend in Form eines Arrays die neuen Werte zu. Dabei spielt es keine Rolle, ob Sie die gleiche Anzahl oder mehr oder weniger Elemente innerhalb des Arrays angeben. Die im Subscript definierten Elemente werden einfach durch die übergebenen neuen Werte ersetzt. Wie das Ganze aussehen kann, zeigt Listing 4.33. Dort werden die ersten zwei Elemente (`Thomas` und `Michaela`) durch drei neue (`Bubi`, `Fine` und `Stephen`) ersetzt, womit das Array am Ende mehr Elemente besitzt als zuvor.

Listing 4.33 Ersetzen mehrerer Elemente eines Arrays

```
names[0...1] = ["Bubi", "Fine", "Stephen"]
print("Namen: \(names)")
// Namen: ["Bubi", "Fine", "Stephen", "Luisa"]
```

4.5.10 Alle Elemente eines Arrays auslesen und durchlaufen

Die `for-in`-Schleife kann im Zusammenspiel mit Arrays genutzt werden, um jedes Element eines Arrays auszulesen und einem temporären Platzhalter zuzuweisen. Für jedes Element des Arrays kann so eine `for-in`-Schleife durchlaufen und der darin deklarierte Code ausgeführt werden.

Welchen Namen Sie dabei für den Platzhalter verwenden, bleibt Ihnen überlassen. In Listing 4.34 sehen Sie ein Beispiel, in dem alle Elemente eines Arrays mit einer `for-in`-Schleife durchlaufen und mittels eines `print()`-Befehls ausgegeben werden.

Listing 4.34 Durchlaufen aller Elemente eines Arrays mittels `for-in`

```
var names = ["Thomas", "Michaela", "Tobias", "Luisa"]
for name in names {
    print("\(name)")
}
// Thomas
// Michaela
// Tobias
// Luisa
```

Statt des Wertebereichs übergeben Sie in diesem Fall der `for-in`-Schleife das zu durchlaufende Array, womit automatisch die Schleife für jedes Element innerhalb des Arrays einmal ausgeführt wird. Das jeweilige Element wird pro Schleifendurchlauf dem temporären Platzhalter zugewiesen, in diesem Beispiel ist dieser mit `name` deklariert. Damit kann die `for-in`-Schleife im Zusammenspiel mit Arrays ideal genutzt werden, um schnell und einfach alle Elemente eines Arrays auf einmal auszulesen und mit jedem eine gewünschte Aktion auszuführen.

Möchten Sie an dieser Stelle zusätzlich noch den jeweiligen Index des Arrays neben dem eigentlichen Wert erhalten, müssen Sie für den Wertebereich der `for-in`-Schleife die Funktion `enumerated()` des Arrays angeben. Diese liefert ebenfalls alle Elemente des Arrays zurück, enthält zusätzlich aber noch den jeweiligen Index als Integer. Damit Sie beide Informationen innerhalb der Schleife auswerten können, müssen Sie auch den Platzhalter entsprechend aktualisieren. Dazu vergeben Sie zwei Bezeichner innerhalb runder Klammern, die durch ein Komma voneinander getrennt werden. Der erste Bezeichner stellt den Platzhalter für den Index dar, der zweite den für den eigentlichen Wert. Listing 4.35 zeigt die Verwendung dieser Funktion und die Auswertung der erhaltenen Informationen pro Schleifendurchlauf.

Listing 4.35 Durchlaufen aller Indexe und Elemente eines Arrays mittels `forin` und `enumerated()`

```
for (index, name) in names.enumerated() {
    print("Index \(index): \(name)")
}
// Index 0: Thomas
// Index 1: Michaela
// Index 2: Tobias
// Index 3: Luisa
```

Tuples

Bei dem in Listing 4.35 definierten Platzhalter (`index, name`) handelt es sich um ein sogenanntes *Tuple*. Tuples erlauben es, mehrere Werte, auch unterschiedlicher Typen, zusammenzufassen. In diesem Beispiel vereint es einen Integer (den Index) mit einem String (dem zugehörigen Namen). Mehr zu Tuples erfahren Sie in Abschnitt 4.8, „Tuple“.

4.6 Set

Der Typ `Set` aus der Swift Standard Library ist vergleichbar mit dem zuvor vorgestellten Typ `Array`. Auch ein Set ist dazu gedacht, mehrere Elemente eines bestimmten Typs zu halten und zu verwalten, allerdings gibt es zwei wichtige Unterschiede zu Arrays. So sind einerseits Sets *nicht sortiert*. Das bedeutet, dass es nicht möglich ist, mithilfe eines Index auf die Elemente eines Sets zuzugreifen, da auf diese Art und Weise womöglich immer ein anderes Element zurückgeliefert werden würde, selbst wenn der verwendete Index immer derselbe ist. Außerdem können Sets *keine doppelten Elemente* enthalten. Wenn Sie einem Set beispielsweise ein Element hinzufügen, das es bereits besitzt, wird nur eine Kopie davon behalten, nicht mehrere.

4.6.1 Erstellen eines Sets

Um ein neues Set zu erstellen, müssen Sie immer die explizite Deklaration mithilfe des Typs `Set` vornehmen. Eine Kurzschreibweise wie bei Arrays gibt es für Sets nicht.

Genau wie Arrays, so sind auch Sets in Swift typsicher. Das bedeutet, dass sie nur Elemente eines bestimmten Typs enthalten können. Das können beispielsweise Strings oder Integer sein, aber keine Mischung aus beiden oder anderen verschiedenen Typen. Zu diesem Zweck wird ein Set mithilfe der Syntax `Set<Element>` deklariert, wobei `Element` für den gewünschten Typ steht, dem die Elemente innerhalb des Sets entsprechen.

Die Werte eines Sets werden genauso deklariert wie die eines Arrays. Das bedeutet, dass Sie die gewünschten Elemente eines Sets innerhalb eckiger Klammern kommasepariert voneinander aufführen.

Listing 4.36 zeigt beispielhaft die Erstellung eines Sets mit verschiedenen Namen. Essenziell ist dabei die konkrete Zuweisung des Typs mittels Type Annotation, da ohne diese Swift stattdessen ein Array erstellen würde.

Listing 4.36 Erstellen eines Sets mit Standardwert

```
var names: Set<String> = ["Thomas", "Michaela", "Tobias"]
```

Sie können in dem gezeigten Beispiel auf die konkrete Nennung des Set-Typs `String` auch verzichten, da dieser anhand der zugewiesenen Werte bereits automatisch für Swift ersichtlich ist. Alternativ zu dem Befehl aus Listing 4.36 können Sie zum Erstellen eines Sets auch den in Listing 4.37 gezeigten Befehl verwenden; das Ergebnis ist bei beiden identisch.

Listing 4.37 Alternative zum Erstellen eines Sets mit Standardwert

```
var names: Set = ["Thomas", "Michaela", "Tobias"]
```

Um ein neues leeres Set zu erstellen, gibt es in Swift zwei Möglichkeiten: Entweder nutzen Sie die Initializer-Syntax des Typs `Set` (mehr zur Initialisierung in Swift erfahren Sie in Kapitel 8, „Initialisierung“) oder Sie weisen einer neuen Variablen beziehungsweise Konstanten ein leeres Set zu. Bei Letzterem gilt es wieder darauf zu achten, den vollständigen Typ mittels Type Annotation anzugeben, da dieser mittels Type Inference von Swift nicht ermittelt werden kann. In Listing 4.38 sehen Sie je ein passendes Beispiel für beide Vorgehensweisen.

Listing 4.38 Erstellen eines neuen leeren Sets

```
// Initializer Syntax
let initializedIntSet = Set<Int>()
// Zuweisen eines leeren Sets mit Type Annotation
let emptyIntegerSet: Set<Int> = []
```

4.6.2 Inhalte eines bestehenden Sets leeren

Es gibt zwei Möglichkeiten, um alle Elemente aus einem Set zu entfernen. Dazu weisen Sie dem Set entweder einen leeren Wert zu (sprich ein eckiges Klammernpaar) oder Sie rufen die Funktion `removeAll()` auf dem entsprechenden Set auf (beide Vorgehensweisen zeigt Listing 4.39). Dabei ist zu beachten, dass dieses Vorgehen nur dann funktioniert, wenn das Set einer Variablen zugewiesen und somit mutable (sprich veränderbar) ist. Auf eine Konstante kann keines der beiden Verfahren angewendet werden.

Listing 4.39 Leeren eines bestehenden Sets

```
var names: Set = ["Thomas", "Michaela", "Tobias"]
// Möglichkeit 1 zum Leeren eines Sets
names = []
// Möglichkeit 2 zum Leeren eines Sets
names.removeAll()
```

4.6.3 Prüfen, ob ein Set leer ist

Mithilfe der Eigenschaft `isEmpty` können Sie überprüfen, ob ein Set über Elemente verfügt oder stattdessen komplett leer ist. Sie erhalten als Ergebnis einen booleschen Wert, der `true` ist, sollte das Set keine Elemente besitzen; andernfalls ist der Wert `false`.

In Listing 4.40 sehen Sie ein kleines Beispiel dazu, das beide Möglichkeiten darstellt und die Verwendung von `isEmpty` im Zusammenspiel mit einer `if`-Abfrage verwendet.

Listing 4.40 Prüfen, ob ein Set leer ist

```
let emptySet = Set<String>()
let notEmptySet: Set = ["Not", "Empty"]
if emptySet.isEmpty {
    print("emptySet ist leer.")
}
if notEmptySet.isEmpty {
    print("notEmptySet ist leer.")
}
// emptySet ist leer.
```

4.6.4 Anzahl der Elemente eines Sets zählen

Sie können die Eigenschaft `count` des Typs `Set` verwenden, um die Anzahl der Elemente eines Sets zu zählen. Als Ergebnis erhalten Sie einen Integer mit der Anzahl der Werte des entsprechenden Sets. Listing 4.41 zeigt ein Beispiel dazu.

Listing 4.41 Zählen der Elemente eines Sets

```
var names: Set = ["Thomas", "Michaela", "Tobias"]
print("names enthält \(names.count) Werte.")
// names enthält 3 Werte.
```

4.6.5 Element zu einem Set hinzufügen

Wenn ein Set als Variable deklariert und somit mutable ist, können diesem jederzeit weitere Elemente hinzugefügt werden. Zu diesem Zweck kommt die Funktion `insert(_:)` zum Einsatz. Diese erwartet als Parameter den Wert, der dem Set hinzugefügt werden soll. Wichtig ist dabei, dass dieser Wert dem passenden Element-Typ des Sets entspricht, andernfalls kommt es zu einem Compiler-Fehler.

Listing 4.42 zeigt beispielhaft die Verwendung der Funktion `insert(_:)`, indem dem Set `names` ein neuer Eintrag hinzugefügt wird.

Listing 4.42 Hinzufügen eines Elements zu einem Set

```
var names: Set = ["Thomas", "Michaela", "Tobias"]
names.insert("Luisa")
print("names: \(names)")
// names: ["Luisa", "Thomas", "Tobias", "Michaela"]
```

4.6.6 Element aus einem Set entfernen

Aus einem mutable Set, das als Variable deklariert ist, können mithilfe der Funktion `remove(_:)` bestehende Elemente wieder entfernt werden. Dabei wird der Funktion das gewünschte zu entfernende Element als Parameter übergeben. Sollte das zugrunde liegende Set kein solches Element enthalten, passiert schlicht nichts, Sie müssen also keinen Absturz Ihrer Anwendung befürchten.

Listing 4.43 demonstriert die mögliche Verwendung der `remove(_:)`-Funktion und entfernt aus einem Set von Namen eines der Elemente.

Listing 4.43 Entfernen eines Elements aus einem Set

```
var names: Set = ["Thomas", "Michaela", "Tobias", "Luisa"]
names.remove("Luisa")
print("names: \(names)")
// names: ["Thomas", "Tobias", "Michaela"]
```

4.6.7 Prüfen, ob ein bestimmtes Element in einem Set vorhanden ist

Um zu überprüfen, ob ein Set ein bestimmtes Element enthält, kann die Funktion `contains(_:)` verwendet werden. Diese gibt den booleschen Wert `true` zurück, wenn das betreffende Set das als Parameter übergebene Element enthält, andernfalls gibt sie `false` zurück. Listing 4.44 zeigt ein Beispiel dazu, in dem die Funktion `contains(_:)` im Zusammenspiel mit einer `if`-Abfrage verwendet wird.

Listing 4.44 Prüfen, ob ein Element in einem Set vorhanden ist

```
var names: Set = ["Thomas", "Michaela", "Tobias", "Luisa"]
if names.contains("Luisa") {
    print("names enthält Luisa.")
} else {
    print("names enthält Luisa nicht.")
}
// names enthält Luisa.
```

4.6.8 Alle Elemente eines Sets auslesen und durchlaufen

Mithilfe einer `for-in`-Schleife ist es möglich, alle Elemente eines Sets nacheinander auszulesen und für jedes dieser Elemente den Code innerhalb der Schleife auszuführen. `forin` ist daher bestens dazu geeignet, um bestimmte Befehle nacheinander auf allen Elementen eines Sets auszuführen.

Zu diesem Zweck wird der `for-in`-Schleife als Wertebereich das Set übergeben, das für jedes seiner Elemente einmal durchlaufen werden soll. Dem selbst definierten Platzhalter der `forin`-Schleife wird automatisch bei jedem Schleifendurchlauf das jeweilige Element aus dem Set zugewiesen. In Listing 4.45 sehen Sie, wie auf diese Art und Weise alle Namen des Sets `names` ausgelesen und innerhalb der `for-in`-Schleife mittels `print()` ausgegeben werden.

Listing 4.45 Durchlaufen aller Elemente eines Sets mittels `for-in`

```
var names: Set = ["Thomas", "Michaela", "Tobias"]
for name in names {
    print("\(name)")
}
// Thomas
// Tobias
// Michaela
```

An dieser Stelle wird auch deutlich, dass ein Set im Vergleich zu einem Array unsortiert ist. Die Reihenfolge, in der die Elemente somit innerhalb der `for-in`-Schleife durchlaufen werden, kann sich bei jedem erneuten Durchlauf ändern.

4.6.9 Sets miteinander vergleichen

Ob zwei Sets über die gleichen Elemente verfügen, können Sie einfach mithilfe des Vergleichsoperators `==` überprüfen. Dieser liefert den booleschen Wert `true` zurück, sollten beide Sets die gleichen Elemente besitzen (wobei die Reihenfolge der Elemente selbstredend keine Rolle spielt), andernfalls `false`. In Listing 4.46 sehen Sie ein Beispiel für solch einen Vergleich mitsamt passendem Ergebnis.

Listing 4.46 Vergleichen von Sets

```
let firstNameSet: Set = ["Michaela", "Tobias", "Thomas"]
let secondNameSet: Set = ["Thomas", "Michaela", "Tobias"]
if firstNameSet == secondNameSet {
    print("firstNameSet und secondNameSet enthalten die gleichen Namen.")
}
// firstNameSet und secondNameSet enthalten die gleichen Namen.
```

Daneben verfügt der Typ `Set` aber noch über weitere Funktionen, die zum Vergleichen zweier Sets verwendet werden können. Auch diese liefern alle einen booleschen Wert zurück. Im Folgenden stelle ich Ihnen all diese Funktionen im Detail vor.

isSubset(of:) und isStrictSubset(of:)

Mittels der Funktion `isSubset(of:)` überprüfen Sie, ob die Elemente eines Sets innerhalb eines anderen Sets enthalten sind. Listing 4.47 zeigt ein Beispiel dazu.

Listing 4.47 Vergleichen von Sets mittels `isSubset(of:)`

```
var firstValues: Set = [19, 99]
var secondValues: Set = [9, 11, 19, 88, 99, 111]
if firstValues.isSubset(of: secondValues) {
    print("secondValues enthält die Elemente von firstValues.")
}
// secondValues enthält die Elemente von firstValues.
```

Die Bedingung in diesem Listing wäre auch dann erfüllt, wenn `firstValues` und `secondValues` exakt die gleichen Elemente beinhalten würden; schließlich ist die Bedingung von `isSubset(of:)` dann noch immer erfüllt, da alle Elemente aus `firstValues` ebenfalls in `secondValues` enthalten sind.

Wenn Sie stattdessen prüfen möchten, ob ein Set alle Elemente eines anderen Sets enthält, beide Sets aber gleichzeitig nicht vollkommen identisch sind, so steht Ihnen dafür die Funktion `isStrictSubset(of:)` zur Verfügung, die genau diese Bedingung prüft. Die Funktion liefert nur dann `true` zurück, wenn einerseits die Elemente eines Sets in einem anderen enthalten sind, gleichzeitig aber beide Sets nicht komplett identisch sind. Listing 4.48 zeigt dazu ein ergänzendes Beispiel.

Listing 4.48 Vergleichen von Sets mittels `isStrictSubset(of:)`

```
var firstValues: Set = [19, 99]
var secondValues: Set = [9, 11, 19, 88, 99, 111]
var thirdValues: Set = [19, 99]
if firstValues.isStrictSubset(of: secondValues) {
    print("secondValues enthält die Elemente von firstValues, die beiden Sets sind
aber nicht identisch.")
}
if firstValues.isSubset(of: thirdValues) {
    print("thirdValues enthält die Elemente von firstValues.")
}
if firstValues.isStrictSubset(of: thirdValues) {
    print("thirdValues enthält die Elemente von firstValues, die beiden Sets sind
aber nicht identisch.")
}
// secondValues enthält die Elemente von firstValues, die beiden Sets sind aber nicht
identisch.
// thirdValues enthält die Elemente von firstValues.
```

Dieses Listing verdeutlicht, das zwar `firstValues` ein Subset von `thirdValues` ist, gleichzeitig aber die Funktion `isStrictSubset(of:)` bei diesem Vergleich `false` zurückliefert, da beide Sets identisch sind.