Inhaltsverzeichnis

TEIL II App-Programmierung

TEIL IV Beispielprojekte

Vorwort

Apples größte Innovation des Jahres 2014 war aus meiner Sicht weder die Vorstellung der Apple Watch noch die Auslieferung des Bestsellers iPhone 6. Apple hat sich neben den Arbeiten an diesen Produkten einer anderen Baustelle zugewandt und als Reaktion auf die vielen Mängel, die die rund 20 Jahre alte Programmiersprache Objective-C aufweist, eine vollkommen neue Programmiersprache entwickelt: Swift!

In ersten Kommentaren konnten selbst Apple-Fans ihre Skepsis nicht verbergen: Brauchen wir wirklich eine neue Programmiersprache? Doch je mehr Details Apple auf der World Wide Developers Conference (WWDC 2014) verriet, desto größer wurde die Begeisterung der teilnehmenden Entwickler und der Fachpresse.

Warum Swift?

Swift ist für Apple ein Befreiungsschlag: Objective-C dient dem Apple-Universum seit vielen Jahren als Fundament. Das ändert aber nichts daran, dass Objective-C eine Programmiersprache aus den 1980er-Jahren ist, die in keinerlei Hinsicht mit modernen Programmiersprachen mithalten kann.

Swift ist dagegen ein sauberer Neuanfang. Bei der Vorstellung wurde Swift auch *Objective-C without the C* genannt. Natürlich ist Swift von Objective-C beeinflusst – schließlich muss Swift kompatibel zu den unzähligen Apple-Bibliotheken sein. Swift realisiert viele neue Ideen, greift aber auch Konzepte von C#, Haskell, Java, Python und anderen Programmiersprachen auf. Daraus ergeben sich mehrere Vorteile:

- Swift zählt zu den modernsten Programmiersprachen, die es momentan gibt.
- Code lässt sich in Swift syntaktisch eleganter formulieren als in Objective-C.
- Der resultierende Code ist besser lesbar und wartbar.
- Swift ist für Programmierer, die schon Erfahrung mit anderen modernen Sprachen gesammelt haben, wesentlich leichter zu erlernen als Objective-C. Vorhandenes Know-how lässt sich einfacher auf Swift als auf Objective-C übertragen.
- Seit Ende 2015 ist Swift ein Open-Source-Produkt. Swift steht seither auch für Linux zur Verfügung, und der Entwicklungsprozess erfolgt offen und transparent.

Wer mit Apple-Produkten zu tun hat, erwartet Perfektion bis ins kleinste Detail. Bei aller Euphorie für Swift will ich Ihnen nicht verschweigen, dass dies für Swift auch in Version 3 nicht vollständig zutrifft:

- Die Integration in Xcode ist gut, aber es ist noch Luft nach oben. Beispielsweise fehlen in Xcode Refactoring-Funktionen für Swift.
- Das Swift-Entwicklerteam versucht, Swift zur optimalen Programmiersprache zu machen. Das geht nicht von heute auf morgen. Es ist absehbar, dass Swift sich in den nächsten Jahren weiterentwickeln wird. Das ist gut so, aber es führt dazu, dass Sie jeden Herbst Ihren Code an die dann gerade neueste Swift-Version anpassen müssen. Die meiste Arbeit erledigt der in Xcode integrierte Konverter automatisch. Aber dennoch wäre mehr Stabilität natürlich angenehmer.

Allen Kinderkrankheiten zum Trotz vereinfacht Swift den Einstieg in die App-Entwicklung enorm. Es ist offensichtlich, dass Swift in wenigen Jahren *die* Programmiersprache der Apple-Welt sein wird und Objective-C in dieser Rolle ablöst. In naher Zukunft führt an Swift kein Weg vorbei, wenn Sie native Apps für iOS, macOS oder andere Apple-Plattformen entwickeln möchten.

Der Swift-Entwicklungsprozess

Seit Herbst 2015 ist Swift ein Open-Source-Projekt. Die Weiterentwicklung erfolgt vollkommen transparent. Jede neue Idee wird als Proposal veröffentlicht, mit ihren Vor- und Nachteilen diskutiert und kommt schließlich zu einer Abstimmung. Nichtsdestotrotz wird beim Nachlesen der Mailing-Archive schnell klar, dass die Entwicklung von Swift trotz des öffentlichen Charakters sehr stark von Apple-Mitarbeitern gesteuert wird.

Ich befasse mich in diesem Buch selten mit der Frage, warum Swift so ist, wie es ist, sondern beschreibe es in der gerade aktuellen Form. Aber wenn Sie Interesse an den Hintergründen haben und mitverfolgen möchten, wie Swift sich entwickelt, lege ich Ihnen das Studium der folgenden Seiten nahe:

- *https://github.com/apple/swift-evolution*
- *https://lists.swift.org/pipermail/swift-evolution*
- *http://ericasadun.com*

Neu in Swift 3

Im Vergleich zu Swift 2 wurden in Version 3 rund hundert Veränderungen durchgeführt. Zu jeder dieser Veränderungen gibt es einen ausführlichen Vorschlag. Die Abschaffung der Operatoren ++ und -- ist beispielsweise im Proposal 0004 dokumentiert. Links zu diesem und allen anderen Proposals finden Sie auf folgender Seite:

https://apple.github.io/swift-evolution

Es ist hier nicht zielführend, alle Änderungen aufzuzählen. Wenn Sie neu in Swift einsteigen, ist es ohnedies irrelevant, wie Swift in früheren Versionen ausgesehen

hat. Und wenn Sie bisher in Swift 2 programmiert haben, kümmert sich der Code-Konverter von Xcode um den Großteil der erforderlichen Änderungen.

Aus meiner Sicht hat sich Swift 3 in den folgenden Punkten radikal verbessert:

- **Bibliotheken:** Die Standardbibliotheken sind wesentlich Swift-kompatibler geworden. Viele Methoden, die ursprünglich für Objective-C entwickelt wurden, stehen nun in einer zweiten, Swift-tauglicheren und besser lesbaren Version zur Verfügung. Hier sehen Sie einige Beispiele, die Code aus Swift 2 aktuellem Swift-3-Code gegenüberstellen:

```
NSUserDefaults.standardUserDefaults()  -->
UserDefaults.standard

zeichenkette.rangeOfString("such mich") -->
zeichenkette.range(of: "such mich")

func prepareForSegue(segue: UIStoryboardSegue,
                     sender: AnyObject?)  -->
func prepare(for segue: UIStoryboardSegue, sender: Any?)

let dest = segue.destinationViewController  -->
let dest = segue.destination

txt.stringByTrimmingCharactersInSet(
   NSCharacterSet.whitespaceAndNewlineCharacterSet()) -->
txt.trimmingCharacters(in: .whitespacesAndNewlines)
```

- **Parameter von Methoden:** In Swift 2 wurde der erste Parameter einer Funktion oder Methode anders behandelt als alle weiteren Parameter. In Swift 3 gelten für alle Parameter dieselben Regeln. Da kann man nur fragen: Warum nicht gleich so?
- **Literale für Farben und Bitmaps:** Farben und Bitmaps aus Xcassets-Dateien können im Code als Literale eingebaut werden. Diese Literale zeigen die Farbe bzw. eine winzige Vorschau der Bitmap. In diesem Buch lässt sich das schwer abbilden, aber in der Praxis ist das ein ausgesprochen hilfreiches Feature.
- **Generics-Verbesserungen:** Beim Umgang mit generischen Typen gab es viele Verbesserungen. Unter anderem können Sie nun generische Aliasse definieren (z. B. `typealias StringDictionary<T> = Dictionary<String, T>`).

Daneben gibt es viele Detailveränderungen, die aus einem Apple-typischen Perfektionsanspruch resultieren. Sie sind spannend für Entwickler von (Standard-)Bibliotheken, aber für den typischen App-Entwickler nur begrenzt relevant.

Keine Angst vor Swift 4!

Es ist schon ein wenig absurd: Im August 2016, also rund drei Monate, bevor dieses Buch veröffentlicht wurde, begannen bereits die Planungen und erste Arbeiten an Swift 4. Zu den wichtigsten Zielen zählt eine Optimierung der generischen Funktionen. Daneben sollen der Umgang mit Zeichenketten und die Speicherverwaltung verbessert werden. Soweit es der enge Zeitplan zulässt, wird es auch kleinere Spracherweiterungen und fortgesetzte Swift-Anpassungen der Standardbibliotheken geben. Schließlich sollen der Compiler und andere Werkzeuge so gestaltet werden, dass mit Swift 3 entwickelte Bibliotheken unverändert unter Swift 4 weiterverwendet werden können. Derart große Umstellungen wie in Swift 3 sollen diesmal aber vermieden werden.

Müssen Sie sich also vor den zu erwartenden Änderungen in Swift 4 fürchten? Ist es zweckmäßig, mit dem Swift-Einstieg auf die nächste Version zu warten? Nein! Bereits der Umstieg von Swift 2 auf Swift 3 hat gezeigt, dass der in Xcode integrierte Code-Konverter einen Großteil aller Änderungen automatisch erledigt.

Außerdem hat sich bei meiner Arbeit an den beiden Auflagen dieses Buchs eines herauskristallisiert: Bei der App-Programmierung kostet nicht der Umgang mit Swift an sich Zeit, sondern die Suche nach den geeigneten Klassen, Methoden und Programmiertechniken.

Sobald Sie Swift einmal in Grundzügen verstanden haben, werden Sie die meiste Zeit dafür investieren, die Dokumentation von Bibliotheken zu studieren bzw. neue Klassen auszuprobieren. Der Kosmos der Apple-Bibliotheken ist derart riesig, dass ich in diesem Buch nicht einmal ansatzweise einen Überblick geben kann. Kurzum: Nicht Swift ist das Problem, sondern die Unmenge an Bibliotheken und Klassen, in die Sie sich einlesen müssen.

Was bietet dieses Buch?

Dieses Buch vermittelt einen kompakten Einstieg in die Programmiersprache Swift in der Version 3 (Xcode 8). Das Buch ist in vier Teile gegliedert:

- **Teil I** führt in die Grundlagen von Swift ein. Hier lernen Sie alle wichtigen Sprachdetails kennen. Die Themenpalette reicht vom Umgang mit Variablen und elementaren Datentypen bis hin zur Syntax der objekt- und protokollorientierten Programmierung.
- **Teil II** ist eine Einführung in die Entwicklung von Apps für iOS, macOS und tvOS. Hier erkläre ich Ihnen beispielsweise, wie der Storyboard-Editor funktioniert, wie Sie Ihre Oberfläche mit eigenem Swift-Code verbinden, eigene `ViewController`-Klassen entwickeln, Tastaturereignisse auswerten oder wie Sie Apps mit mehreren Dialogen/Views organisieren.

- **Teil III** fasst wichtige Programmiertechniken in Bausteinform zusammen. In Kurzanleitungen zeige ich Ihnen unter anderem, wie Sie auf Dateien zugreifen, XML-Dokumente auswerten, Webseiten anzeigen, Steuerelemente mit eigener Grafik gestalten, Listen und Tabellen in Apps darstellen, geografische Daten auswerten und Spiele mit SpriteKit programmieren. Sobald Ihre App mit diesen Techniken zufriedenstellend funktioniert, lernen Sie, wie Sie sie tauglich für den App Store machen und dort einreichen. Der Großteil der hier präsentierten Techniken funktioniert plattformübergreifend, also gleichermaßen in iOS-, tvOS- und macOS-Apps.
- **Teil IV** zeigt anhand konkreter Beispielprojekte die Praxis der App-Programmierung. Die Apps decken eine ganze Palette von Themen ab: vom praktischen Währungsumrechner über den Icon-Resizer bis hin zu mehreren Spielen.

Selbstverständlich können Sie alle Beispieldateien und -projekte dieses Buchs herunterladen. Einen Download-Link finden Sie hier:

www.rheinwerk-verlag.de/swift_4115

Um von diesem Buch maximal zu profitieren, benötigen Sie weder Vorkenntnisse in Xcode noch in der App-Entwicklung. Ich setze aber voraus, dass Sie bereits Erfahrungen mit einer beliebigen Programmiersprache gesammelt haben. Ich erkläre Ihnen in diesem Buch also, wie Sie in Swift mit Variablen umgehen, Schleifen programmieren und Klassen entwickeln, aber nicht, was Variablen sind, wozu Schleifen dienen und warum Klassen das Fundament der objektorientierten Programmierung sind. So kann ich Swift kompakt und ohne viel Overhead beschreiben und den Schwerpunkt auf die konkrete Anwendung legen.

Leseanleitung

Fast 1200 Seiten – das kann schon abschrecken! Dazu besteht aber kein Grund. Ich habe mich beim Schreiben dieses Buchs bemüht, den Inhalt auf möglichst eigenständige Kapitel zu verteilen, aus denen Sie sich wie aus einem Baukasten bedienen können.

Wenn Swift für Sie vollständig neu ist, dann ist die Lektüre der ersten Kapitel aus Teil I natürlich unumgänglich. Besonders wichtig ist, dass Sie die Swift-spezifischen Eigenheiten beim Umgang mit elementaren Datentypen und Aufzählungen (Arrays, Dictionaries etc.) kennenlernen und das Konzept von Optionals verstehen. Interessanterweise hat sich herausgestellt, dass Sie für die Entwicklung erster Apps nicht unbedingt alle Feinheiten im Zusammenhang mit Vererbung, Protokollen etc. wissen müssen. Die Basics reichen zumeist.

Teil II richtet sich speziell an Programmierer, die erstmalig Apps für iOS, macOS oder watchOS entwickeln. Wenn Sie bisher Objective-C zur App-Programmierung verwendet haben, werden Sie in Teil II auf viel bekanntes Wissen stoßen.

Bei Teil III habe ich versucht, oft benötigte Programmiertechniken möglichst losgelöst von der Zielplattform zu beschreiben. Beispielsweise erfolgt der Umgang mit Dateien unter iOS ganz ähnlich wie unter macOS. Aus den Kapiteln in Teil III können Sie sich also bedienen, wie Sie es gerade brauchen.

Viele Detailprobleme treten erst dann auf, wenn man den Schritt von kleinen Beispielen hin zu »richtigen« Apps macht. Deswegen stellt Teil IV eine Reihe vollständiger Projekte vor. Auch wenn es unwahrscheinlich ist, dass Sie genau so eine App programmieren möchten wie eines der Beispiele aus Teil IV, so werden Sie in diesen Kapiteln vermutlich doch inhaltlich verwandte Anleitungen und Arbeitstechniken mit einem hohen Praxisbezug finden. Probieren Sie die Apps einfach einmal aus, und blättern Sie dann durch die entsprechenden Kapitel – Sie werden sicher über Details stolpern, die sich später als hilfreich herausstellen werden.

Viel Spaß bei der App-Entwicklung!

Eine neue Programmiersprache zu erlernen ist immer eine Herausforderung. Noch schwieriger ist es, einen Überblick über die schier unüberschaubare Fülle von Bibliotheken zu gewinnen, die Sie zur App-Entwicklung brauchen. Dieses Buch soll Ihnen bei beiden Aspekten helfen und Ihnen ein solides Fundament vermitteln.

Wenn Sie in die App-Programmierung mit Swift einsteigen, haben Sie das Privileg, mit einer der modernsten aktuell verfügbaren Programmiersprachen arbeiten zu können. Sobald Sie die ersten Schritte einmal erfolgreich absolviert haben, wird die Faszination für diese Sprache auch Sie erfassen. Bei Ihrer Reise durch die neue Welt der Swift-Programmierung wünsche ich Ihnen viel Spaß und Erfolg!

Michael Kofler (*https://kofler.info*)

PS: Ausdrücklich bedanken möchte ich mich bei Alfred Schilken, der eine Rohfassung des Manuskript gelesen und mich auf viele Ungenauigkeiten hingewiesen hat.

Kapitel 3
Operatoren

Im Ausdruck `a = b + c` gelten die Zeichen `=` und `+` als Operatoren. Dieses Kapitel stellt Ihnen alle Swift-Operatoren vor – von den simplen Operatoren für die Grundrechenarten bis hin zu Swift-Spezialitäten wie dem Range-Operator `n1...n2`.

Leerzeichen vor oder nach Operatoren

Normalerweise ist es nicht notwendig, vor oder nach einem Operator ein Leerzeichen zu schreiben. `x=x+7` funktioniert genauso gut wie `x = x + 7`. Aber wie so oft bestätigen Ausnahmen die Regel: Swift kennt bereits standardmäßig ungewöhnlich viele Operatoren, und wenige Zeilen Code reichen aus, um weitere zu definieren.

Das führt mitunter dazu, dass der Compiler nicht eindeutig erkennen kann, wo der eine Operator endet und wo der nächste beginnt. Spätestens dann *müssen* Sie ein Leerzeichen setzen – und dann sollten Sie es vor *und* nach dem Operator setzen! Andernfalls glaubt der Compiler nämlich, Sie wollten ihn explizit darauf hinweisen, dass es sich um einen Präfix- oder Postfix-Operator handelt. Im Detail sind diese Feinheiten in »The Swift Programming Language« dokumentiert:

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/LexicalStructure.html (beim Punkt »Operators«)

Wie rasch Probleme auftreten können, zeigt der Vergleich eines Optionals mit `nil`: Der Compiler betrachtet `if opt!=nil` als Syntaxfehler, weil er die Anweisung als `if opt! = nil` interpretiert, also als Kombination zweier Operatoren. Korrekt müssen Sie den Vergleich in der Form `if opt != nil` formulieren.

3.1 Zuweisungs- und Rechenoperatoren

Dieser Abschnitt erläutert die zahlreichen Rechen- und Zuweisungsoperatoren. Swift kennt dabei auch Mischformen. Beispielsweise entspricht `x+=3` der Anweisung `x=x+3`.

Einfache Zuweisung

Der Zuweisungsoperator = speichert in einer Variablen oder Konstanten das Ergebnis des Ausdrucks:

```
variable = ausdruck
```

Vor der ersten Zuweisung an eine Variable bzw. Konstante muss diese mit `var` bzw. `let` als solche deklariert werden:

```
var i = 17
i = i * 2
let pi = 3.1415927
```

Nicht zulässig sind Mehrfachzuweisungen in der Art `a=b=3`. Dafür können mehrere Variablen als Tupel geschrieben und gleichzeitig verändert werden:

```
var (a, b, c) = (1, 7, 12)
```

Das funktioniert auch bei komplexeren Ausdrücken:

```
var (_, a, (b, c)) = (1, 2, ("x", "y"))
// entspricht var a=2; var b="x"; var c="y"
```

Der Unterstrich _ ist hier ein *Wildcard Pattern*. Es trifft auf jeden Ausdruck zu und verhindert im obigen Beispiel dessen weitere Verarbeitung.

Wert- versus Referenztypen

Swift unterscheidet bei Zuweisungen zwischen zwei grundlegenden Datentypen:

- **Werttypen (Value Types):** Dazu zählen Zahlen, Zeichenketten, Tupel, Arrays, Dictionaries sowie `struct`- und `enum`-Daten. Bei einer Zuweisung werden die Daten kopiert. Die ursprünglichen Daten und die Kopie sind vollkommen unabhängig voneinander.
- **Referenztypen:** Objekte, also Instanzen von Klassen, sind Referenztypen. Bei einer Zuweisung wird eine weitere Referenz auf die bereits vorhandenen Daten erstellt. Es zeigen nun zwei (oder mehr) Variablen auf dieselben Daten.

Die folgenden beiden Beispiele verdeutlichen den Unterschied. Im ersten Beispiel werden in `x` und `y` ganze Zahlen gespeichert, also Werttypen:

```
var x = 3
var y = x
x=4
print(y)  // y ist unverändert 3
```

Für das zweite Beispiel definieren wir zuerst die Mini-Klasse `SimpleClass`. In `a` wird eine Instanz dieser Klasse gespeichert. Bei der Zuweisung `b = a` wird die Instanz *nicht*

kopiert, stattdessen verweist nun b auf dasselbe Objekt wie a. (In C würde man sagen, a und b sind Zeiger.) Jede Veränderung des Objekts betrifft deswegen a gleichermaßen wie b:

```
class SimpleClass {
  var data=0
}

var a = SimpleClass()
var b = a         // a und b zeigen auf die gleichen Daten
a.data = 17
print(b.data)     // deswegen ist auch b.data 17
```

Arrays, Dictionaries und Zeichenketten sind Werttypen!

Die Unterscheidung zwischen Wert- und Referenztypen gibt es bei den meisten Programmiersprachen. Beachten Sie aber, dass Arrays und Zeichenketten in Swift Werttypen sind und nicht, wie in vielen anderen Sprachen, Referenztypen!

Elementare Rechenoperatoren

Die meisten Rechenoperatoren sind aus dem täglichen Leben bekannt (siehe Tabelle 3.1). Der Operator % liefert den Rest einer ganzzahligen Division: `13 % 5` ergibt also 3, da `2 * 5 + 3 = 13`. Bei Fließkommazahlen wird der Rest zum ganzzahligen Ergebnis ermittelt. `1.0 % 0.4` ergibt `0.2`, da `2 * 0.4 + 0.2 = 1.0` ist.

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Restwert einer ganzzahligen Division
&+	Integer-Addition ohne Überlaufkontrolle
&-	Integer-Subtraktion ohne Überlaufkontrolle
&*	Integer-Multiplikation ohne Überlaufkontrolle

Tabelle 3.1 Rechenoperatoren

Alle Operatoren setzen voraus, dass links und rechts von ihnen jeweils gleichartige Datentypen verwendet werden! Im Gegensatz zu anderen Programmiersprachen erfolgen Typumwandlungen nicht automatisch.

```
var a = 3        // a ist eine Integer-Variable
var b = 1.7      // b ist eine Fließkommavariable
var c = a + b    // Fehler, Int-Wert + Double-Wert nicht zulässig
```

Wenn Sie die Summe von `a` plus `b` ausrechnen möchten, müssen Sie explizit den Datentyp einer der beiden Operatoren anpassen. `Int` rundet dabei immer ab, d. h., aus `1.7` wird `1`.

```
var c1 = a + Int(b)      // c1 = 4
var c2 = Double(a) + b   // c2 = 4.7
```

Division durch null

Bei einer Fließkommadivision durch 0.0 lautet das Ergebnis einfach `Double.infinity` bzw. `-Double.infinity`. Wenn Sie hingegen mit Integerzahlen arbeiten, löst eine Division durch 0 einen Fehler aus.

Eine Besonderheit von Swift sind die Operatoren `&+`, `&-` und `&*`: Sie führen die Grundrechenarten für Integerzahlen ohne Überlaufkontrolle durch. Das ermöglicht die Programmierung besonders effizienter Algorithmen. Sollte allerdings doch ein Überlauf eintreten, dann ist das Ergebnis falsch!

```
var i = 10000000           // Integer
var result = i &* i &* i   // falsches Ergebnis
                           // 3.875.820.019.684.212.736
```

Swift kennt keinen Operator zum Potenzieren. a^b müssen Sie unter Zuhilfenahme der Funktion `pow` berechnen. Diese Funktion ist in der `Foundation`-Bibliothek definiert. Sie steht nur zur Verfügung, wenn Ihr Code `import Foundation` enthält oder eine andere Bibliothek importiert, die auf die `Foundation` zurückgreift. Das trifft unter anderem für Cocoa und UIKit zu.

```
var e = 7.0
var f = pow(e, 3.0)  // 7 * 7 * 7 = 343.0
```

Zeichenketten aneinanderfügen

Der Operator `+` addiert nicht nur zwei Zahlen, sondern fügt auch Zeichenketten aneinander:

```
var s1 = "Hello"
var s2 = "World!"
var hw = s1 + " " + s2  // "Hello World!"
```

Inkrement und Dekrement

Wie viele andere Programmiersprachen kannte Swift bis zur Version 2 die Inkrement- und Dekrement-Operatoren ++ und --. In Swift 3 wurden diese Operatoren allerdings eliminiert, mit der Begründung, dass die Unterscheidung zwischen der Postfix- und der Präfix-Notation, also zwischen `i++` und `++i`, zu viel Verwirrung und oft fehlerhafte Algorithmen verursacht.

Wenn Sie in Swift 3 eine Variable um eins vergrößern oder verkleinern möchten, müssen Sie dies in der Form `i=i+1` oder in der Kurzschreibweise `i+=1` machen.

```
var i=3
i++     // Fehler, steht in Swift 3 nicht mehr zur Verfügung
i+=1    // OK
```

Inkrement- und Dekrement-Operatoren selbst gemacht

Wenn Sie auf die Inkrement- und Dekrement-Operatoren nicht verzichten möchten, bietet Swift die Möglichkeit, diese Operatoren relativ unkompliziert wieder selbst zu definieren. Fertigen Code für alle Integer-Datentypen inklusive `Int8`, `Int16`, `Int32` und `Int64` sowie alle `UInt`-Varianten finden Sie z. B. hier auf Github:

https://gist.github.com/erica/6b4be87de789f32b8926388c6c6e75e9

Rechnen mit Bits

Die bitweisen Operatoren &, |, ^ und ~ (AND, OR, XOR und NOT) verarbeiten ganze Zahlen bitweise. Das folgende Beispiel verwendet die Schreibweise `0b` zur Kennzeichnung binärer Zahlen. `String` mit dem zusätzlichen Parameter `radix:2` wandelt ganze Zahlen in eine Zeichenkette in binärer Darstellung um.

```
let a = 0b11100                    // Wert 28
let b = 0b01111                    // Wert 15
let result = a & b                 // Wert 12
print(String(result, radix:2))     // Ausgabe 1100
```

>> verschiebt die Bits einer Zahl um *n* Bits nach rechts (entspricht einer Division durch 2^n), << verschiebt entsprechend nach links (entspricht einer Multiplikation mit 2^n). >>> funktioniert wie >>, betrachtet die Zahl aber so, als wäre sie vorzeichenlos.

```
let e = 16
let f = e << 2  // entspricht f=e*4, Ergebnis 64
let g = e >> 1  // entspricht g=e/2, Ergebnis 8
```

Wenn Sie Daten bitweise verarbeiten, ist es oft zweckmäßig, anstelle gewöhnlicher Integer-Zahlen explizit Datentypen ohne Vorzeichen zu verwenden, z. B. `UInt32` oder `UInt16`. Das folgende Beispiel verwendet `0x` zur Kennzeichnung hexadezimaler Zahlen.

```
let rgb: UInt32 = 0x336688
let red: UInt8  = UInt8( (rgb & 0xff0000) >> 16 )
```

Kombinierte Rechen- und Zuweisungsoperationen

Alle bereits erwähnten Rechenoperatoren sowie die logischen Operatoren && und || können mit einer Zuweisung kombiniert werden. Dazu muss dem Operator das Zeichen = folgen. Details zu den logischen Operatoren folgen im nächsten Abschnitt.

```
x += y     // entspricht x = x + y
x -= y     // entspricht x = x - y
x *= y     // entspricht x = x * y
x /= y     // entspricht x = x / y
x %= y     // entspricht x = x % y
x <<= y    // entspricht x = x << y
x >>= y    // entspricht x = x >> y
x &= y     // entspricht x = x & y
x &&= y    // entspricht x = x && y
x |= y     // entspricht x = x | y
x ||= y    // entspricht x = x || y
x ^= y     // entspricht x = x ^ y
```

3.2 Vergleichsoperatoren und logische Operatoren

Um Bedingungen für Schleifen oder Verzweigungen zu formulieren, müssen Sie Variablen vergleichen und oft mehrere Vergleiche miteinander kombinieren. Dieser Abschnitt stellt Ihnen die dazu erforderlichen Operatoren vor.

Vergleichsoperatoren

Die Vergleichsoperatoren ==, != (ungleich), <, <= sowie > und >= können gleichermaßen für Zahlen und für Zeichenketten eingesetzt werden. Wie bei anderen Operatoren ist es wichtig, dass auf beiden Seiten des Operators der gleiche Datentyp verwendet wird; Sie können also nicht eine ganze Zahl mit einer Fließkommazahl vergleichen!

```
1 == 2          // false
1 < 2           // true

"abc" == "abc"  // true
"abc" == "Abc"  // false
```

Zeichenketten gelten dann als gleich, wenn auch die Groß- und Kleinschreibung übereinstimmt. Etwas schwieriger ist die Interpretation von *größer* und *kleiner*. Grundsätzlich gelten Großbuchstaben als *kleiner* als Kleinbuchstaben, d. h., sie werden beim Sortieren vorne eingereiht. Internationale Zeichen werden auf der Basis der *Unicode-Normalform D* verglichen. Die deutschen Buchstaben ä, ö oder ü werden dabei wie eine Kombination aus zwei Zeichen betrachtet, beispielsweise `ä = a"`. Somit gilt:

```
"A" < "a"       // true
"a" < "ä"       // true
"ä" < "b"       // true
```

Mehr Details zur Sortierordnung von Zeichenketten und zu Möglichkeiten, diese zu beeinflussen, folgen in Kapitel 8, »Zeichenketten«.

== versus ===

Zum Vergleich von Objekten kennt Swift neben `==` und `!=` auch die Varianten `===` und `!==`. Dabei testet `a===b`, ob die beiden Variablen `a` und `b` auf dieselbe Instanz einer Klasse zeigen. Hingegen überprüft `a==b`, ob `a` und `b` zwei Objekte mit übereinstimmenden Daten sind. Das ist nicht das Gleiche! Es ist ja durchaus möglich, dass zwei unterschiedliche Objekte dieselben Daten enthalten.

Einschränkungen

Die Operatoren `===` und `!==` können nur auf Referenztypen angewendet werden, nicht auf Werttypen (wie Zahlen, Zeichenketten, Arrays, Dictionaries sowie sonstige Strukturen).

Umgekehrt können die Operatoren `==` und `!=` bei selbst definierten Klassen nur verwendet werden, wenn Sie für diese Klassen den Operator `==` selbst implementieren (Protokoll `Equatable`, siehe Abschnitt 12.4, »Standardprotokolle«).

Die folgenden Zeilen definieren zuerst die Klasse `Pt` zur Speicherung eines Koordinatenpunkts und dann den Operator `==` zum Vergleich zweier `Pt`-Objekte. Damit ist das Beispiel gleich auch ein Vorgriff auf die Definition eigener Operatoren.

```
class Pt {
    var x: Double, y: Double
    // Init-Funktion
    init(x: Double, y: Double){
        self.x=x
        self.y=y
    }
}

// Operator zum Vergleich von zwei Pt-Objekten
func ==(left: Pt, right: Pt) -> Bool {
    return left.x == right.x && left.y == right.y
}

// == versus ===
var p1 = Pt(x: 1.0, y: 2.0)
var p2 = Pt(x: 1.0, y: 2.0)
p1 == p2    // true, weil die Objekte dieselben Daten enthalten
p1 === p2   // false, weil es unterschiedliche Objekte sind
```

Vergleiche mit ~=

Swift kennt mit `~=` einen weiteren Vergleichsoperator mit recht wenigen Funktionen:

- Zwei Ausdrücke des gleichen Typs werden wie mit `==` verglichen.
- Außerdem kann getestet werden, ob eine ganze Zahl in einem durch den Range-Operator formulierten Zahlenbereich enthalten ist.

  ```
  -2...2 ~= 1   // true
  -2...2 ~= -2  // true
  -2...2 ~= 2   // true
  -2...2 ~= 4   // false
  ```

  Achten Sie darauf, dass Sie zuerst den Bereich und dann den Vergleichswert angeben müssen. Wenn Sie die Reihenfolge vertauschen, funktioniert der Operator nicht. Details zu Range-Operatoren folgen gleich.

Analog kann auch in `switch`-Ausdrücken mit `case` überprüft werden, ob sich ein ganzzahliger Ausdruck in einem vorgegebenen Bereich befindet:

```
let n = 12
switch n {
case (1...10):
  print("Zahl zwischen 1 und 10")

case(11...20):
  print("Zahl zwischen 11 und 20")
```

```
default:
  print("Andere Zahl")
}
```

Datentyp-Vergleich (»is«)

Mit dem Operator `is` testen Sie, ob eine Variable einem bestimmten Typ entspricht:

```
func f(obj: Any) {
  if obj is UInt32 {
    print("Datentyp UInt32")
    ...
  }
}
```

Casting-Operator (»as«)

Mit dem Operator `as` wandeln Sie, sofern möglich, einen Datentyp in einen anderen um. Der Operator hat in Swift drei Erscheinungsformen:

- `as`: In dieser Form eignet sich `as` nur, wenn der Compiler erkennen kann, dass die Umwandlung gefahrlos möglich ist. Das trifft auf alle Upcasts zu, also auf Umwandlungen in Instanzen einer übergeordneten Klasse (siehe Abschnitt 12.1, »Vererbung«), außerdem bei manchen Literalen (z. B. `12 as Float`).
- `as?`: Der Downcast-Operator `as? typ` stellt vor der Typkonvertierung sicher, dass diese überhaupt möglich ist. Wenn das nicht der Fall ist, lautet das Ergebnis `nil`. Es tritt kein Fehler auf.

  Mit `if let varname = ausdruck as? typ` können Sie den Typtest mit einer Zuweisung kombinieren. Das funktioniert gleichermaßen für Konstanten (`let` wie im folgenden Beispiel) wie auch für Variablen (`var`):

  ```
  func f(obj: Any) {
    if let myint = obj as? UInt32 {
      // myint hat den Datentyp UInt32
      ...
    } else {
      print("falscher Datentyp")
    }
  }
  ```

- `as!`: Mit einem nachgestellten Ausrufezeichen wird die Konvertierung auf jeden Fall versucht. Dabei kann es zu einem Fehler kommen, wenn der Datentyp nicht passt. Insofern ist diese Variante zumeist nur zweckmäßig, wenn die Typüberprüfung im Voraus erfolgt.

```
let obj: Any = 123
if obj is UInt32 {
  // wird nicht ausgeführt, weil obj eine Int-Instanz enthält
  var myint = obj as! UInt32
}
```

Upcasts und Downcasts

Ein Grundprinzip der objektorientierten Programmierung ist die Vererbung. Damit können Klassen die Merkmale einer Basisklasse übernehmen und diese erweitern oder verändern. Ein Objekt einer abgeleiteten Klasse (im Klassendiagramm unten dargestellt) kann immer wie ein Objekt der Basisklasse verwendet werden (im Klassendiagramm oben). Das nennt man einen (impliziten) Upcast, also eine Umwandlung in der Klassenhierarchie nach oben.

Eine Konvertierung in die umgekehrte Richtung ist ein Downcast. Dieser funktioniert nur, wenn eine Variable vom Typ der Basisklasse tatsächlich ein Objekt der erforderlichen abgeleiteten Klasse enthält. Mehr Details zu diesem Thema finden Sie in Abschnitt 12.1, »Vererbung«.

Logische Operatoren

Logische Operatoren kombinieren Wahrheitswerte. *Wahr UND Wahr* liefert wieder *Wahr*; *Wahr UND Falsch* ergibt hingegen *Falsch*. In Swift gibt es wie in den meisten anderen Programmiersprachen die drei logischen Operatoren `!` (Nicht), `&&` (Und) sowie `||` (Oder):

```
let a=3, b=5
a>0 && b<=10              // true
a>b || b>a                // true
let ok = (a>b)            // false
if !ok {                  // wenn ok nicht true ist, dann ...
  print("Fehler")         // ... eine Fehlermeldung ausgeben
}
```

`&&` und `||` führen eine sogenannte *Short-Circuit Evaluation* aus: Steht nach der Auswertung des ersten Operanden das Endergebnis bereits fest, wird auf die Auswertung des zweiten Ausdrucks verzichtet. Wenn im folgenden Beispiel `a>b` das Ergebnis `false` liefert, dann ruft Swift die Funktion `calculate` gar nicht auf; der logische Ausdruck ist in jedem Fall `false`, ganz egal, welches Ergebnis `calculate` liefern würde.

```
if a>b && calculate(a, b)==14 { ... }
```

3.3 Range-Operatoren

In Swift gibt es zwei Operatoren, um Bereiche für Zahlen oder Zeichenketten auszudrücken:

- Der Closed-Range-Operator `a...b` beschreibt einen Bereich von `a` bis inklusive `b`.
- Der Half-Open-Range-Operator `a..<b` beschreibt hingegen einen Bereich von `a` bis exklusive `b`, entspricht also `a...b-1`.

Normalerweise werden diese Range-Operatoren für ganze Zahlen verwendet. Es gibt aber auch Anwendungsfälle für Fließkommazahlen und Zeichenketten, auf die ich im nächsten Abschnitt eingehe.

`a` muss kleiner als `b` sein, sonst ist der Ausdruck ungültig. Mit den Range-Operatoren definierte Bereiche können in Schleifen, in `switch-case`-Ausdrücken sowie mit dem vorhin vorgestellten Operator `~=` verarbeitet werden.

```
for i in 1...10 {
    print(i)
}
```

In der folgenden Schreibweise können Sie Zahlenbereiche zur Initialisierung eines Integer-Arrays verwenden:

```
var ar = [Int](1...10)  // entspricht var ar = [1, 2, ..., 10]
```

Mit `map` können Sie auf einen Zahlenbereich direkt eine Funktion (Closure) anwenden:

```
(1...5).map { print($0) }
```

Die Operatoren `...` und `..<` sind in Wirklichkeit Kurzschreibweisen zur Erzeugung von `CountableRange`- bzw. `CountableClosedRange`-Strukturen.

```
1..<10          // entspricht CountableRange<Int>(1..<10)
1...10          // entspricht CountableClosedRange<Int>(1...10)
```

`CountableRange` und `CountableClosedRange` sind also Datenstrukturen, die einen Zahlenbereich repräsentieren. Die wichtigsten Eigenschaften sind `startIndex` und `endIndex`. Sie geben den Start- bzw. Endwert des Bereichs an. `contains` überprüft, ob eine bestimmte Zahl im Zahlenbereich enthalten ist.

```
var r = 1..<10  // entspricht r = CountableRange<Int>(1..<10)
r.startIndex    // 1
r.endIndex      // 10
r.contains(7)   // true
r.contains(11)  // false
```

Darüber hinaus gibt es unzählige Methoden, um den Zahlenbereich zu verändern (`dropFirst`, `dropLast`, `reversed`) bzw. um seine Elemente zu verarbeiten (`filter`, `forEach`, `map`).

Range-Operatoren für Fließkommazahlen und Zeichenketten

Je nach Kontext können die Operatoren `a...b` und `a..<b` auch zur Formulierung eines Intervalls verwendet werden – z. B. als Vergleichsbasis für `switch` bzw. für den schon beschriebenen Operator `~=`. In diesem Fall sind für `a` und `b` auch Fließkommazahlen oder Zeichenketten (`Character`) erlaubt.

```
1...10 ~= 8          // true
1.7..<2.9 ~= 2.3     // true
"a"..."z" ~= "f"     // true
"0"..."9" ~= "x"     // false
```

Allerdings gelten für Bereiche, deren Grunddatentyp nicht aufzählbar ist, viele Einschränkungen. Beispielsweise ist es unmöglich, Schleifen über sie zu bilden:

```
for c in "a"..."z" {    // Fehler, ClosedRange<String> does not
  print(c)              // conform to protocol sequence
}
```

Schleifen über einen Fließkommazahlenbereich

Um analog zu `for i in 1...3` eine Schleife für eine `double`-Variable zu formulieren, können Sie auf `stride` zurückgreifen. Diese Funktion stelle ich Ihnen in Kapitel 5, »Verzweigungen und Schleifen«, vor.

3.4 Operatoren für Fortgeschrittene

Die wichtigsten Operatoren kennen Sie nun. Dieser Abschnitt ergänzt Ihr Wissen um Spezial- und Hintergrundinformationen. Am interessantesten ist dabei sicherlich die Möglichkeit, selbst eigene Operatoren zu definieren bzw. vorhandene Operatoren zu überschreiben (Operator Overloading).

Ternärer Operator

Swift kennt drei Typen von Operatoren:

- **Unäre Operatoren** (Unary Operators) verarbeiten nur einen Operanden. In Swift zählen dazu das positive und negative Vorzeichen, das logische NICHT (also `!bool`) und die Unwrapping-Operatoren `!` und `?` zur Auswertung von Optionals (also `opt!` oder `opt?`).

- **Binäre Operatoren** verarbeiten zwei Operanden, also etwa das `a * b`. Die Mehrheit der Swift-Operatoren fällt in diese Gruppe.
- **Ternäre Operatoren** verarbeiten drei Operanden. In Swift gibt es nur einen derartigen Vertreter, der daher einfach als *der* ternäre Operator bezeichnet wird – so, als wären andere ternäre Operatoren undenkbar.

Die Syntax des ternären Operators sieht so aus:

```
a ? b : c
```

Wenn der boolesche Ausdruck `a` wahr ist, dann liefert der Ausdruck `b`, sonst `c`. Der ternäre Operator eignet sich dazu, einfache `if`-Verzweigungen zu verkürzen:

```
// if-Schreibweise
let result:String
let x = 3
if x<10 {
    result = "x kleiner 10"
} else {
    result = "x größer-gleich 10"
}

// verkürzte Schreibweise mit dem ternären Operator
let x = 3
let result = x<10 ? "x kleiner 10" : "x größer gleich 10"
```

Unwrapping- und Nil-Coalescing-Operator

Anders als in den meisten anderen Programmiersprachen können mit einem Typ deklarierte Swift-Variablen nie den Zustand `null` im Sinne von »nicht initialisiert« annehmen. Swift bietet dafür die Möglichkeit, eine Variable explizit als *Optional* zu deklarieren (siehe Abschnitt 4.2, »Optionals«). Dazu geben Sie explizit den gewünschten Datentyp an, dem wiederum ein Fragezeichen oder ein Ausrufezeichen folgt:

```
var x: Int? = 3     // x enthält eine ganze Zahl oder nil
var y: Int! = 4     // y enthält eine ganze Zahl oder nil
var z: Int  = 5     // z enthält immer eine ganze Zahl

x = nil             // ok
y = nil             // ok
z = nil             // nicht erlaubt
```

Der Unterschied zwischen `x` und `y` besteht darin, dass das Auspacken (*Unwrapping*) des eigentlichen Werts bei `y` automatisch erfolgt, während es bei `x` durch ein nachgestelltes Ausrufezeichen – den Unwrapping-Operator – erzwungen werden muss.

Beachten Sie, dass die folgenden Zeilen beide einen Fehler verursachen, wenn x bzw. y den Zustand nil aufweist!

```
var i: Int = x!      // explizites Unwrapping durch x!
var j: Int = y       // automatisches Unwrapping
```

Die Variablen x und y können also nil enthalten. nil hat eine ähnliche Bedeutung wie bei anderen Sprachen null. Optionals sind aber gleichermaßen für Wert- und für Referenztypen vorgesehen, was in manchen anderen Programmiersprachen nicht oder nur auf Umwegen möglich ist. Allerdings können nur solche Klassen für Optionals verwendet werden, die das Protokoll ExpressibleByNilLiteral einhalten.

Mit diesem Vorwissen kommen wir nun zum Nil-Coalescing-Operator a ?? b, der sich ebenso schwer aussprechen wie übersetzen lässt. Bei ihm handelt es sich um eine Kurzschreibweise des folgenden Ausdrucks:

```
a != nil ? a! : b
```

Wenn a initialisiert ist, also nicht nil ist, dann liefert a ?? b den Wert von a zurück, andernfalls den Wert von b. Damit eignet sich b zur Angabe eines Defaultwerts. In a! bewirkt das Ausrufezeichen das Auspacken (*Unwrapping*) des Optionals. Aus dem Optional Typ? wird also der reguläre Datentyp Typ.

```
// k den Wert von x oder den Defaultwert -1 zuweisen
var k = x ?? -1       // der Datentyp von k ist Int
```

Optional Chaining

Ebenfalls mit Optionals hat die Operatorkombination ?. zu tun. Sie testet, ob ein Ausdruck nil ergibt. Ist dies der Fall, lautet das Endergebnis nil. Andernfalls wird das Ergebnis ausgepackt und der nächste Ausdruck angewendet. Wenn dieser ebenfalls ein Optional liefert, kann auch diesem Ausdruck ein Fragezeichen hintangestellt werden. Swift führt einen weiteren nil-Test durch. Diese Verkettung von nil-Tests samt Auswertung, wenn der Ausdruck nicht nil ist, heißt in Swift »Optional Chaining«:

```
let a = optional?.method()?.property
let b = optional?.method1()?.method2()?.method3()
```

Operator-Präferenz

Beim Ausdruck a + b * c rechnet Swift zuerst b*c aus, bevor es summiert – so wie Sie es in der Schule gelernt haben. Generell gilt in Swift eine klare Hierarchie der Operatoren (siehe Tabelle 3.2). Um die Verarbeitungsreihenfolge zu verändern, können Sie natürlich jederzeit Klammern setzen – also beispielsweise (a+b)*c.

Priorität	Operatoren
BitwiseShiftPrecedence	<< >>
MultiplicationPrecedence ←	* / % &*
AdditionPrecedence ←	+ - &+ &- | ^
RangeFormationPrecedence	... ..<
CastingPrecedence	is as
NilCoalescingPrecedence	??
ComparisonPrecedence	< <= > >= == != === !=== ~=
LogicalConjunctionPrecedence ←	&&
LogicalDisjunctionPrecedence ←	||
DefaultPrecedence	(keine zugeordneten Operatoren)
TernaryPrecedence →	?:
AssignmentPrecedence →	= *= /= %= += -= <<= >>= &= ^= |= &&= ||=
FunctionArrowPrecedence →	->

Tabelle 3.2 Hierarchie der binären Operatoren

In Swift gibt es vordefinierte Prioritätsgruppen (Precedence Groups). Die erste Spalte der Operatorentabelle gibt neben dem Namen der Gruppe auch die Assoziativität an (soweit definiert). Diese bestimmt, ob gleichwertige Operatoren von links nach rechts oder von rechts nach links verarbeitet werden sollen. Beispielsweise ist - (Minus) ein linksassoziativer Operator. Die Auswertung erfolgt von links nach rechts. `17 - 5 - 3` wird also in der Form `(17 - 5)- 3` verarbeitet und ergibt 9. Falsch wäre `17 - (5 - 3) = 15`!

Detailinformationen zu den Prioritätsgruppen sowie weitere Operator-Interna können Sie im Proposal 0077 nachlesen. Es fasst all die Änderungen zusammen, die in Swift 3 im Bereich der Operatoren durchgeführt wurden:

https://github.com/apple/swift-evolution/blob/master/proposals/0077-operator-precedence.md

3.5 Eigene Operatoren

Swift bietet die Möglichkeit, vorhandenen Operatoren für bestimmte Datentypen neue Funktionen zuzuweisen (Operator Overloading). Außerdem können Sie vollkommen neue Operatoren definieren.

Operatoren sind aus der Sicht von Swift ein Sonderfall globaler Funktionen, wobei der Funktionsname aus Operatorzeichen besteht. Insofern greift dieser Abschnitt Kapitel 6, »Funktionen und Closures«, vor. Binäre Operatoren erwarten zwei, unäre Operatoren einen Parameter.

Reservierte Operatoren

Die folgenden Zeichen, Zeichenkombinationen bzw. Operatoren sind reserviert und können nicht verändert werden:

`= -> // /* */ . > ! ?`

Operator Overloading für komplexe Zahlen

Das folgende Beispiel zeigt, wie einfach Operator Overloading ist. Im Beispiel wird zuerst die Datenstruktur `Complex` zur Speicherung komplexer Zahlen mit Real- und Imaginärteil definiert. Die weiteren Zeilen zeigen die Implementierung der Operatoren + und * zur Verarbeitung solcher Zahlen.

```
// Beispieldatei operator-overloading.playground

// Datenstruktur zur Speicherung komplexer Zahlen
struct Complex {
  var re: Double
  var im: Double

  // Init-Funktion, um eine neue komplexe Zahl zu erzeugen
  init(re: Double, im: Double) {
    self.re=re
    self.im=im
  }
}

// Addition komplexer Zahlen
func + (left: Complex, right: Complex) -> Complex {
  return Complex(re: left.re + right.re, im: left.im + right.im)
}
```

```
// Multiplikation komplexer Zahlen
func * (left: Complex, right: Complex) -> Complex {
  return Complex(re: left.re * right.re - left.im * right.im,
                 im: left.re * right.im + left.im * right.re)
}

// Vergleich komplexer Zahlen
func == (left: Complex, right: Complex) -> Bool {
  return left.re == right.re   &&   left.im == right.im
}
func != (left: Complex, right: Complex) -> Bool {
  return !(left==right)
}

// Operatoren anwenden
var a = Complex(re: 2, im: 1)  //  2 +  i
var b = Complex(re: 1, im: 3)  //  1 + 3i
var c = a + b                  //  3 + 4i
var d = a * b                  // -1 + 7i
```

Bei der Definition unärer Operatoren muss mit dem Schlüsselwort `prefix` bzw. `postfix` angegeben werden, ob der Operator vor oder nach dem Operanden angegeben wird. Die Definition des Operators für negative Vorzeichen bei komplexen Zahlen sieht so aus:

```
// negatives Vorzeichen für komplexe Zahlen
prefix func - (op: Complex) -> Complex {
  return Complex(re: -op.re, im: -op.im)
}

// Anwendung
var e = -d                     //  1 - 7i
```

Neuer Vergleichsoperator für Zeichenketten

Für aktuell nicht genutzte Zeichenkombinationen können Sie selbst neue Operatoren definieren. Dazu müssen Sie den Operator mit `infix|prefix|postfix operator` zuerst definieren und einer Prioritätsgruppe zuordnen (siehe Tabelle 3.2). Wenn Sie darauf verzichten, zählt Ihr Operator automatisch zur Gruppe `DefaultPrecedence`.

Das folgende Beispiel definiert einen Vergleichsoperator für Zeichenketten, der nicht zwischen Groß- und Kleinschreibung unterscheidet. `infix` bezeichnet dabei einen Operator für zwei Operanden. Der Operator `=~=` erhält dieselbe Priorität wie die anderen Vergleichsoperatoren.

```
// Datei comparision-operator.swift
// neuen Vergleichsoperator definieren, ...
infix operator =~= : ComparisonPrecedence
```

Die Implementierung greift auf die `compare`-Methode zurück. Besser lesbar, aber weniger effizient wäre `return left.uppercased()== right.uppercased()`.

```
// ... implementieren,
func =~= (left: String, right: String) -> Bool {
  return left.compare(right, options: .caseInsensitive) == .
    orderedSame
}

// ... und ausprobieren
"abc" =~= "Abc"   // true
"äöü" =~= "ÄöÜ"   // true
```

Neuer Exponential-Operator in einer eigenen Prioritätsgruppe

Sie sind bei der Definition eigener Operatoren nicht auf die vorgegebenen Prioritätsgruppen beschränkt. Das folgende Beispiel greift eine Idee aus dem schon erwähnten Proposal 0077 auf und definiert den Exponential-Operator `**` für `Double`-Zahlen. Der Operator wird der ebenso neuen Gruppe `ExponentiationPrecedence` zugeordnet. Diese Gruppe hat eine höhere Priorität als `MultiplicationPrecedence`, mehrere aufeinanderfolgende Exponential-Operatoren werden von links nach rechts verarbeitet.

```
// Datei comparision-operator.swift

// neue Prioritätsgruppe für Exponential-Operatoren
precedencegroup ExponentiationPrecedence {
  associativity: left
  higherThan: MultiplicationPrecedence
}

// neuer Exponential-Operator
infix operator ** : ExponentiationPrecedence

// Implementierung
func ** (base: Double, exponent: Double) -> Double {
  return pow(base, exponent)
}

let result = 1 + 2 ** 3  // 1 plus (2 hoch 3), Ergebnis 9
```

Weitere Beispiele für eigene Operatoren

In diesem Buch zeige ich Ihnen in mehreren Abschnitten, wie Sie eigene Operatoren gewinnbringend zur Formulierung klareren Codes definieren können:

- In Abschnitt 7.3, »CGFloat, CGPoint, CGSize und Co.«, zeige ich Ihnen, wie Sie Operatoren definieren, um bequem Berechnungen mit Koordinatenpunkten, Vektoren etc. durchzuführen.
- In Abschnitt 8.5, »Zeichenketten und Zahlen umwandeln«, stelle ich Ihnen einen Dreizeiler vor, der den aus Python bekannten Operator % zur Formatierung von Zeichenketten definiert.
- In Abschnitt 11.5, »Methoden«, habe ich in einem Beispiel, das den Unterschied zwischen verschiedenen Methodentypen zeigt, auch die Operatoren < und > definiert. Sie ermitteln, welcher von zwei 3D-Vektoren kürzer ist.

Kapitel 15
Hello iOS-World!

Mit diesem Kapitel beginnt nun endlich die konkrete iOS-Programmierung. In seinem Zentrum steht ein einfaches Hello-World-Programm. Das Ziel ist es, Sie mit den Grundfunktionen von Xcode sowie mit einigen Konzepten der iOS-Programmentwicklung bekannt zu machen.

Bei der Lektüre dieses und auch des nächsten Kapitels werden Sie feststellen, dass Swift vorübergehend in den Hintergrund rückt. Natürlich müssen wir da und dort ein paar Zeilen Code schreiben; aber gerade bei den ersten iOS-Programmen geht es vielmehr darum, den effizienten Umgang mit Xcode zu erlernen, die grafische Gestaltung von Apps kennenzulernen und grundlegende Konzepte der iOS-Programmentwicklung zu verstehen. Dennoch setze ich ab diesem Kapitel voraus, dass Sie zumindest über elementare Grundkenntnisse in Swift verfügen. Insbesondere die grundlegenden Sprachstrukturen, Datentypen und der Umgang mit Optionals sollte Ihnen geläufig sein.

Xcode per Video kennenlernen

Ich wäre nicht seit 30 Jahren mit Begeisterung Autor, würde ich nicht aus vollster Überzeugung das Medium Buch lieben. Aber es heißt ja, ein Bild sagt mehr als tausend Worte, und man könnte diese Analogie noch fortführen: Ein Video sagt mehr als tausend Bilder. Mitunter stimmt das sogar, z. B. wenn es darum geht, die Bedienung von Xcode zu erlernen.

Ohne Sie hier an die Konkurrenz verweisen zu wollen, möchte ich Ihnen doch ein Video-Tutorial ans Herz legen, das qualitativ meilenweit über den vielen YouTube-Filmchen der Art »Jetzt lerne ich iOS-Programmierung« steht. Die renommierte Stanford Universität hat eine rund 20-stündige Vorlesung zum Thema »Developing iOS 8 Apps with Swift« kostenlos im iTunes-U-Programm verfügbar gemacht. Um die Videos ansehen zu können, benötigen Sie entweder iTunes auf Ihrem Mac oder die App *iTunesU* für das iPad. Grundlegende Englischkenntnisse reichen aus, um dem mit vielen Screencasts unterlegten Vortrag zu folgen. Die Videos basieren momentan auf Swift 2.0, es ist aber zu hoffen, dass es Anfang 2017 eine aktualisierte Fassung gibt, die iOS 10 und vor allem Swift 3 berücksichtigt.

https://itunes.apple.com/de/course/developing-ios-9-apps-swift/id1104579961

15.1 Projektstart

Um eine App für ein iOS-Gerät zu entwickeln, also für ein iPhone, ein iPad oder einen iPod, starten Sie in Xcode ein neues Projekt und wählen zuerst den Typ iOS • Single View Application. Das ist die einfachste Art von iOS-Apps. Anfänglich besteht eine derartige App aus einer einzigen Ansicht, also aus einem »Bildschirm«, aus einer »Seite«. Sie können aber später problemlos weitere Ansichten hinzufügen. Der anfängliche Projekttyp schränkt Sie also nicht bei der späteren Weiterentwicklung der App ein.

Im nächsten Dialogblatt geben Sie Ihrem Projekt einen Namen und wählen als Programmiersprache Swift sowie als Device Universal aus. Das bedeutet, dass Sie die App später sowohl auf einem iPhone als auch auf einem iPad ausführen können.

Der Dialog enthält drei Optionen (siehe Abbildung 15.1), die Sie allesamt deaktiviert lassen können und die in diesem Buch nicht weiter behandelt werden:

- Use Core Data ist nur dann relevant, wenn Sie die Daten Ihrer App in einem Objekt-Relationen-Mapping-Modell speichern möchten.
- Include Unit Tests und Include UI Tests fügen Ihrem Projekt jeweils ein eigenes Verzeichnis hinzu, in dem Sie Testfunktionen für Ihren Code bzw. für die Benutzeroberfläche unterbringen können.

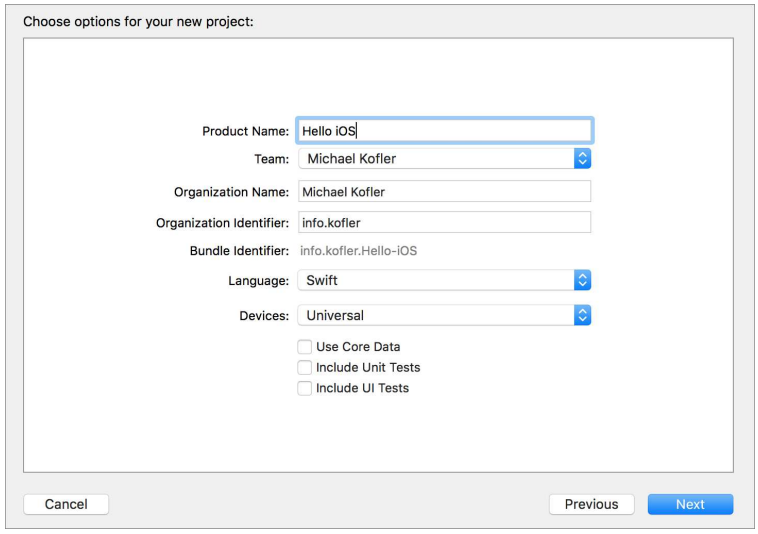

Abbildung 15.1 Projektoptionen für das Hello-iOS-Projekt

Zuletzt müssen Sie noch das Verzeichnis angeben, in dem Xcode die Code-Dateien des neuen Projekts speichern soll.

15.2 Gestaltung der App

Das neue Projekt besteht aus einer ganzen Reihe von Dateien, deren Namen in der linken Spalte von Xcode im Projektnavigator zu sehen sind. Sollte Xcode den Projektnavigator nicht anzeigen, führen Sie VIEWS • NAVIGATOR • PROJECT NAVIGATOR aus oder drücken einfach ⌘+1. Im Projektnavigator klicken Sie nun auf die Datei `Main.storyboard`.

Im Hauptbereich von Xcode sehen Sie jetzt das sogenannte *Storyboard* (siehe Abbildung 15.2). Anfänglich besteht dieses Drehbuch für iOS-Apps aus nur einer, vorerst leeren Szene, also einer Bildschirmansicht der App. Bei vielen Apps gesellen sich dazu später weitere Seiten, zwischen denen die Anwender dann navigieren können. Im Hello-World-Beispiel wird es aber bei einer Ansicht bleiben.

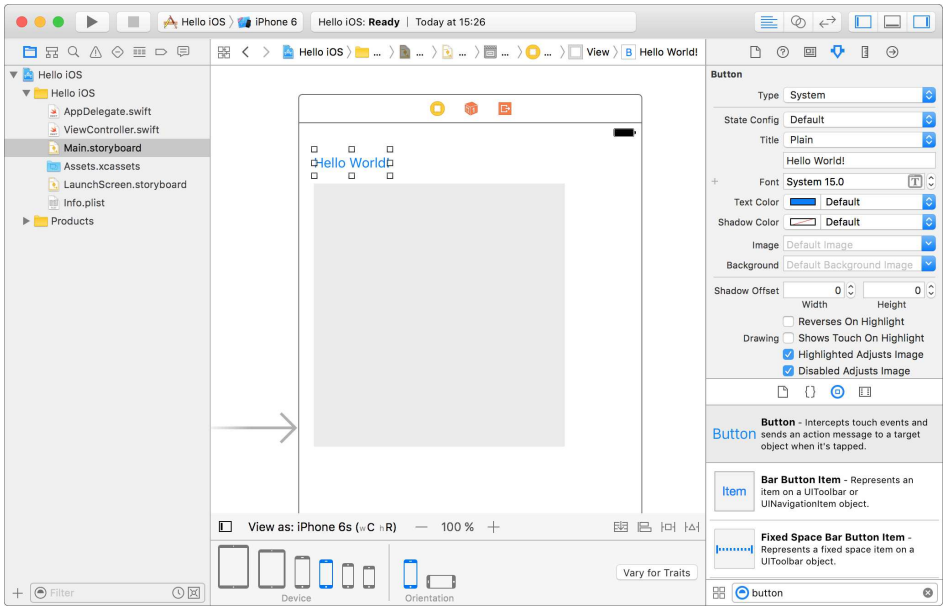

Abbildung 15.2 Das Storyboard mit der einzigen Programmansicht der Hello-World-App

Device-Größe

Bei neuen iOS-Apps zeigt Xcode die Ansichten der App im Storyboard so an, wie diese auf einem iPhone 6 im Hochformat aussehen würden. Wenn Sie in der Fußleiste des Storyboard-Editors auf VIEW AS IPHONE 6S klicken, blendet Xcode eine Auswahl von iOS-Geräten mit anderen Bildschirmformaten ein. Per Knopfdruck können Sie nun zwischen einem dieser Formate wählen (siehe Abbildung 15.2).

Dieses Feature ist neu in Xcode 8 und ermöglicht es Ihnen, blitzschnell zu überprüfen, wie Ihre App auf unterschiedlichen Geräten aussehen wird. Anfänglich wird Ihre App nicht auf die unterschiedlichen Bildschirmproportionen und -maße reagieren. Abhilfe schaffen sogenannte Layout-Regeln, die ich Ihnen in Abschnitt 15.5, »Layout optimieren«, und detaillierter nochmals in Abschnitt 16.5, »Auto Layout«, vorstellen werde.

Ich verwende bei der Entwicklung meiner Apps meistens die Device-Größe »iPhone SE«. Das spart einerseits wertvollen Platz im Storyboard-Editor und gibt mir andererseits eine gute Vorstellung davon, wie die App auf dem kleinsten infrage kommenden Gerät für iOS 10 aussehen wird. In der Regel ist die Optimierung für kleine Bildschirme schwieriger als die für große. (Wenn Sie in den Projekteigenschaften das Deployment Target auf iOS 9.n zurücksetzen, können Sie auch noch Apps für das iPhone 4S entwickeln.)

Mini-Glossar

Bevor Sie damit beginnen, die Hello-World-App zu gestalten, sollten Sie zumindest die wichtigsten Grundbegriffe aus der iOS-Entwicklerwelt kennen: Eine App kann aus mehreren Ansichten bestehen. Solche Ansichten nennt Xcode *Szenen*. Jede Szene kann diverse Steuerelemente enthalten, die das Aussehen der Ansicht bestimmen. Alle Einstellungen aller Szenen sowie der darin enthaltenen Steuerelemente werden im *Storyboard* gespeichert.

Jede Szene wird mit einer Code-Datei verbunden, dem *Controller*. Im Controller legen Sie fest, wie Ihre App auf Ereignisse reagiert. Der Controller bestimmt also, wie sich Ihre App verhalten soll, wenn ein Benutzer einen Button berührt, einen Slider verschiebt etc. Das dem Controller zugrunde liegende Programmiermuster, den *Model-View-Controller*, stelle ich Ihnen in Abschnitt 16.1 näher vor.

Standardmäßig enthält ein neues Projekt vom Typ Single View Application die Storyboard-Datei `Main.storyboard` mit der Szene »View-Controller« und dem zugeordneten Controller in der Datei `ViewController.swift`. Um später weitere Szenen hinzuzufügen, verschieben Sie einen View-Controller aus der Xcode-Objektbibliothek in das Storyboard. Während im Storyboard beliebig viele Szenen gespeichert werden können, benötigen Sie im Regelfall zu jeder Szene eine eigene Swift-Code-Datei mit dem Controller. Mitunter können Sie auch eine Controller-Klasse mehreren ähnlichen Szenen zuordnen. Wie Sie selbst neue Controller-Klassen einrichten, beschreibe ich in Abschnitt 17.1, »Storyboard und Controller-Klassen verbinden«.

Vorsicht bei Umbenennungen!

Die grafische Darstellung des Storyboards in Xcode ist sehr ansprechend. Intern handelt es sich beim Storyboard aber um eine simple XML-Datei. Dabei ist es wichtig zu wissen, dass die Referenzen vom Storyboard auf Code-Elemente und Komponenten ausschließlich in Form von Zeichenketten erfolgen.

Oft ist die Versuchung groß, Outlets, Actions, Klassennamen etc. nachträglich zu verändern, wenn sich die ursprünglich gewählten Namen als irreführend oder zu wenig prägnant erweisen. Damit geraten Sie aber schnell in Teufels Küche. Xcode beklagt sich dann über unbekannte Objekte in den Interface-Builder-Dateien, die aus dem Storyboard generiert werden. Abhilfe schafft das Neueinfügen bzw. Neuverbinden von Actions und Outlets bzw. die Neuzuordnung von Klassen; mitunter ist es aber gerade für Xcode-Einsteiger schwierig, die wahre Ursache des Problems zu finden.

Steuerelemente einfügen

Die Benutzeroberfläche unserer Hello-World-App ist anfänglich leer. In die Ansicht sollen nun zwei Bestandteile (Steuerelemente) eingebaut werden: Ein Button mit der Aufschrift Hello World! und ein leeres Textfeld. Jedes Mal, wenn Sie den Button Hello World! anklicken, soll im Textfeld eine neue Zeile mit dem aktuellen Datum und der Uhrzeit eingefügt werden (siehe Abbildung 15.3).

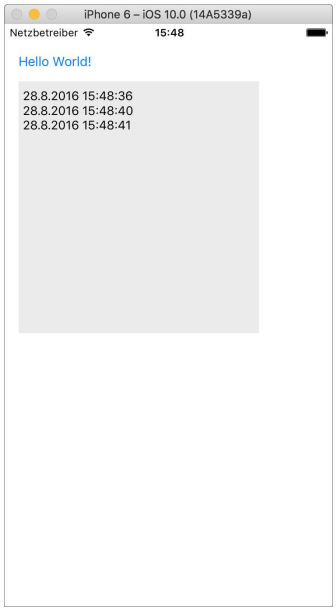

Abbildung 15.3 Die Hello-World-App im iOS-Simulator

Xcode zeigt die zur Auswahl stehenden Steuerelemente normalerweise rechts unten in der Objektbibliothek an. Sollten die Steuerelemente bei Ihnen nicht sichtbar sein, blenden Sie diesen Bereich der Werkzeugleiste mit VIEW • UTILITIES • SHOW OBJECT LIBRARY bzw. mit ctrl+alt+⌘+3 ein.

Xcode kann die Objektbibliothek in einer kompakten Icon-Ansicht oder in einer Listenansicht darstellen. Wechseln Sie gegebenenfalls mit dem Button am unteren Rand der Objektbibliothek in die übersichtlichere Listenansicht. Im daneben befindlichen Suchfeld können Sie in der Liste rasch die gewünschten Einträge ermitteln – suchen Sie nach *button* bzw. *text view*.

Klicken Sie zuerst den Button an, und verschieben Sie ihn per Drag & Drop in den quadratischen View-Controller des Storyboards. Platzieren Sie den Button links oben, und ändern Sie dann den Text von BUTTON zu HELLO WORLD (siehe Abbildung 15.2). Wenn Sie möchten, können Sie im Attributinspektor links oben in der Werkzeugleiste die Button-Schrift etwas größer einstellen. Den Attributinspektor blenden Sie bei Bedarf mit alt+⌘+4 ein.

Nun ist das Text-View-Steuerelement an der Reihe: Auch dieses verschieben Sie per Drag & Drop aus der Objektbibliothek in das Storyboard. Platzieren Sie das Steuerelement unterhalb des Buttons, und stellen Sie die Größe so ein, dass es circa ein Viertel des quadratischen Storyboards füllt. Wie gesagt, mit den Feinheiten des Layouts beschäftigen wir uns später. Stellen Sie nun die gewünschte Textgröße ein, und löschen Sie den Blindtext »Lorem ipsum … « im Attributinspektor.

Wenn Sie möchten, können Sie dem Textfeld auch noch eine andere Hintergrundfarbe zuweisen – dann erkennen Sie später bei der Programmausführung besser die tatsächlichen Ausmaße des Steuerelements. Zur Farbeinstellung scrollen Sie im Attributinspektor nach unten, bis Sie im Bereich VIEW das Feld BACKGROUND finden.

Ein erster Test mit dem iOS-Simulator

Nachdem wir nun die minimalistische Oberfläche unserer App gestaltet haben, spricht nichts dagegen, das Programm ein erstes Mal zu starten. Dazu wählen Sie in der Symbolleiste des Xcode-Fensters zuerst das Gerät aus, das Sie simulieren möchten. Zur Auswahl steht die gängige iOS-Hardware-Palette.

Ein Klick auf den dreieckigen RUN-Button kompiliert das Programm und übergibt es zur Ausführung an den iOS-Simulator (siehe Abbildung 15.3). Dabei handelt es sich um ein eigenständiges Programm, das getrennt von Xcode läuft. Wie der Name vermuten lässt, simuliert es ein iPhone bzw. iPad, wobei das zu testende Programm zur Ausführung automatisch installiert wird. Der Simulator ist zumindest für erste Tests gut geeignet. Für »echte« Apps führt an Tests auf richtiger Hardware aber natürlich kein Weg vorbei.

Spracheinstellungen im Simulator

Im Simulator gelten anfänglich englische Spracheinstellungen. Das lässt sich wie auf einem richtigen Gerät in den Systemeinstellungen beheben. Dazu beenden Sie die laufende Hello-World-App durch einen per Menü mit HARDWARE • HOME simulierten Druck auf den iPhone-Button. Anschließend starten Sie die vorinstallierte App SETTINGS, navigieren in den Dialog GENERAL • LANGUAGE & REGION und stellen dort die Sprache DEUTSCH und die Region GERMANY ein.

Diesen Vorgang müssen Sie für jedes iOS-Gerät wiederholen, das Sie testen möchten. Die Einstellungen werden also nicht über alle simulierten Geräte geteilt.

Da die Veränderung der Spracheinstellungen in allen iOS-Geräten recht umständlich ist, bietet Xcode auch die Möglichkeit, mit PRODUCT • SCHEME • EDIT SCHEME festzulegen, in welcher Sprache die App ausgeführt werden soll. Das Hello-World-Kapitel ist aber nicht der richtige Ort für diese schon recht fortgeschrittenen Funktionen. Details zum Umgang mit Schemes und zur Entwicklung mehrsprachiger Apps folgen in Abschnitt 29.4.

Im iOS-Simulator können Sie nun den Button HELLO WORLD! anklicken. Das Programm wird darauf aber nicht reagieren, weil der Button ja noch nicht mit Code verbunden ist. Wenn Sie das Textfeld anklicken, können Sie über die Tastatur Ihres Computers sowie über die im Simulator eingeblendete Bildschirmtastatur Text eingeben. Die Test-App läuft, bis Sie sie in Xcode mit dem Stopp-Button beenden.

Ärger mit der Tastatur

Sie können im iOS-Simulator mit der Tastatur Ihres Computers Eingaben durchführen. Das ist effizient, aber nicht realitätsnah: Der Simulator betrachtet Ihre Tastatur als mit dem iOS-Gerät verbunden und blendet deswegen die iOS-Bildschirmtastatur aus. Mit dem Kommando HARDWARE • KEYBOARD • CONNECT HARDWARE KEYBOARD im iOS-Simulator können Sie auswählen, ob Sie Ihre Computertastatur oder die Bildschirmtastatur von iOS verwenden möchten.

Sobald die iOS-Bildschirmtastatur einmal im iOS-Simulator erscheint, tritt ein weiteres Problem auf: Es scheint keine Möglichkeit zu geben, die Tastatur wieder loszuwerden. Das ist das in iOS vorgesehene Verhalten – wenn die Eingabe abgeschlossen ist, muss die Tastatur per Code explizit ausgeblendet werden, in der Regel durch die Anweisung `view.endEditing(true)`. Im Hello-World-Projekt verzichten wir darauf.

Praktische Details zum richtigen Umgang mit Textfeldern folgen in Abschnitt 16.7, »Texteingaben«, sowie in mehreren Beispielprojekten in Teil IV dieses Buchs.

15.3 Steuerung der App durch Code

Damit das Berühren bzw. im Simulator das Anklicken des Buttons eine sichtbare Wirkung zeigt, soll die App jetzt so erweitert werden, dass in das Textfeld jedes Mal eine Zeile mit dem Datum und der aktuellen Uhrzeit hinzugefügt wird. Das erfordert einige Zeilen eigenen Swift-Code.

Den Button mit einer Methode verbinden (Actions)

Mit der App-Ansicht (*Szene*) verbundener Code muss nun in die bereits vorgesehene Datei `ViewController.swift` eingetragen werden. Xcode unterstützt diesen Vorgang, sofern das Storyboard und die damit verbundene Datei `ViewController.swift` nebeneinander angezeigt werden. Genau das bewirkt der Button mit den »Eheringen«: Er blendet zur gerade aktiven Datei den sogenannten Assistenzeditor ein, also eine zugeordnete zweite Datei.

Platz sparen in Xcode

Wenn Sie nicht gerade auf einem sehr großen Monitor arbeiten, wird der Platz in Xcode jetzt knapp. Links sitzt der Projektnavigator, in der Mitte das Storyboard und der Code des Controllers und rechts noch die Werkzeugleiste – das ist zu viel! Blenden Sie deswegen vorübergehend die beiden Seitenleisten aus. Die entsprechenden Buttons finden Sie rechts oben in der Xcode-Symbolleiste.

`ViewController.swift` enthält den Swift-Code für eine Klasse mit dem Namen `ViewController`. Diese Klasse ist von der `UIViewController`-Klasse abgeleitet. Wie der Name vermuten lässt, steuert diese Klasse das App-Erscheinungsbild. Der View-Controller enthält bereits zwei leere Methoden. Diese sind für uns vorerst aber nicht von Interesse; wenn Sie möchten, können Sie sie löschen. In diesem Hello-World-Beispiel benötigen wir sie nicht.

Dafür möchten wir nun eine Methode in den Code einbauen, die ausgeführt wird, wenn der Button HELLO WORLD angeklickt wird. Dazu drücken Sie die Taste ctrl und ziehen den Button vom Storyboard in den Assistenzeditor. Die Methode soll direkt in der Klasse und *nicht* innerhalb einer anderen Methode eingefügt werden. Achten Sie darauf, die Maus- bzw. Trackpad-Taste an der richtigen Stelle loszulassen! Bis dahin markiert eine blaue Linie die Verbindung, die Sie herstellen möchten.

Sobald Sie die Maus bzw. das Trackpad losgelassen haben, erscheint ein Dialog, in dem Sie die Details der Verbindung einstellen können (siehe Abbildung 15.4). Für uns sind folgende Einstellungen zweckmäßig:

- CONNECTION = ACTION: Wir wollen auf ein Ereignis reagieren. Der Begriff »Action« bezieht sich dabei auf das Target-Action-Entwurfsmuster, bei dem eine Methode in die eigene Controller-Klasse eingebaut wird. Sie gilt als »Ziel«, das angesprungen wird, wenn das entsprechende Ereignis auftritt.
- NAME: Hier geben wir der zu erstellenden Methode einen möglichst aussagekräftigen Namen. Ich habe mich bei diesem Beispiel für `hwButtonTouch` entschieden. Wenn Sie das Beispiel selbst ausprobieren, können Sie hier einen beliebigen anderen Namen wählen.
- TYPE = ANYOBJECT oder UIBUTTON: Diese Einstellung gibt an, in welcher Form Daten an die Methode übergeben werden sollen. Im Hello-World-Beispiel werten wir diese Daten aber ohnehin nicht aus – insofern ist die Einstellung egal. Wollten wir den Text des Buttons oder andere Eigenschaften auslesen, wäre es zweckmäßig, hier UIBUTTON einzustellen.
- EVENT = TOUCH UP INSIDE: Hier wird festgelegt, auf welches Ereignis wir reagieren möchten. Die Defaulteinstellung passt hier: Wenn der Button innen berührt wird, soll unser Code ausgeführt werden.
- ARGUMENTS = SENDER: In der Auswahlliste können Sie festlegen, welche Daten an die Methode übergeben werden sollen. Zur Auswahl stehen SENDER, SENDER AND EVENT oder NONE. Die Einstellung SENDER bedeutet, dass eine Referenz auf das Objekt übergeben wird, das die Aktion ausgelöst hat. Das vereinfacht die Auswertung der Daten dieses Objekts – insbesondere dann, wenn Sie eine Action-Methode mit mehreren Steuerelementen verbinden.

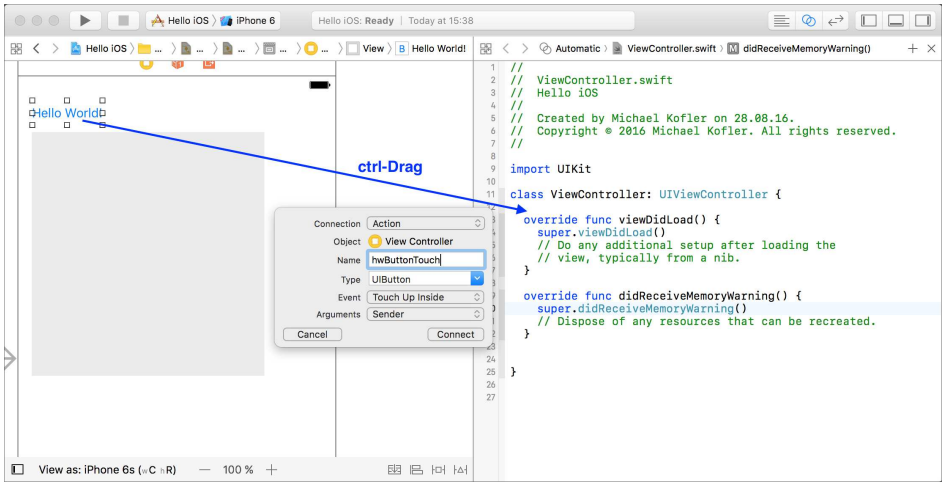

Abbildung 15.4 Der Button des Storyboards (links) wird durch ctrl-Drag mit einer neuen Methode im ViewController-Code (rechts) verbunden.

Die hier so langatmig beschriebenen Einstellungen nehmen Sie mit etwas Übung in gerade mal 15 Sekunden vor. Xcode belohnt diese Arbeit mit einer Methodendefinition, die aus zwei Zeilen Code besteht:

```
// Projekt hwios-world, Datei ViewController.swift
@IBAction func hwButtonTouch(_ sender: UIButton) {
  // todo: Code verfassen
}
```

Der Code allein reicht nicht!

Angesichts des doch recht umständlichen Mausgeklickes liegt es nahe, den wenigen Code einfach selbst einzutippen. Das ist aber keine gute Idee! Mit dem Code an sich ist es nämlich nicht getan. Hinter den Kulissen merkt sich Xcode auch, mit welchem Steuerelement und mit welchem Ereignis die Methode verknüpft ist. Diese Metadaten landen in der XML-Datei `Main.storyboard`. Die Verbindung stellt Cocoa Touch dynamisch zur Laufzeit her. Das kompilierte Storyboard enthält dafür eine Referenz auf den serialisierten View-Controller und den Selektor auf die Methode.

Zugriff auf das Textfeld über eine Eigenschaft (Outlets)

Bevor wir die Methode `hwButtonTouch` mit konkretem Code füllen können, benötigen wir noch eine Möglichkeit, um auf das Textfeld zuzugreifen. Dazu stellen wir eine weitere Verbindung zwischen der App-Ansicht im Storyboard und dem Code her: Wieder mit ctrl ziehen wir diesmal das Text-View-Steuerelement in den Codebereich, verwenden diesmal aber andere Einstellungen (siehe Abbildung 15.5):

- Connection = Outlet: Wir wollen auf das Steuerelement zugreifen können. Xcode soll also eine entsprechende Eigenschaft in die `ViewController`-Klasse einbauen, die auf das Objekt verweist. In der iOS-Nomenklatur wird das als *Outlet* bezeichnet.
- Name: Das ist der Name, unter dem wir das Steuerelement ansprechen möchten. Ich habe `textView` angegeben.
- Type = UITextView: Das Steuerelement ist eine Instanz der `UITextView`-Klasse – und unter diesem Typ möchten wir auf das Steuerelement auch zugreifen.
- Storage = Weak: Hier geht es darum, ob die Referenz auf das Steuerelement sicherstellt, dass dieses im Speicher bleibt. Das ist nicht notwendig: Das Steuerelement kommt uns sicher nicht abhanden. Insofern können wir die Defaulteinstellung Weak bedenkenlos übernehmen.

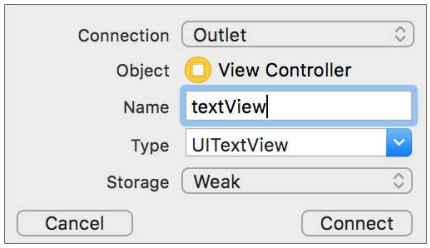

Abbildung 15.5 Einstellungen für das textView-Outlet

Die resultierende Code-Zeile sieht so aus:

```
@IBOutlet weak var textView: UITextView!
```

Damit ist also eine neue Eigenschaft (Klassenvariable) mit dem Namen `textView` definiert. Wir können damit auf ein Steuerelement vom Typ `UITextView` zugreifen, wobei der Typ als Optional angegeben ist. Der Grund dafür ist, dass die Initialisierung des Steuerelements möglicherweise nicht sofort beim App-Start, sondern erst etwas später erfolgt. Indem die Eigenschaft als Optional deklariert ist, wird klar, dass ein Zugriff auf das Steuerelement unter Umständen noch gar nicht möglich ist.

Die Attribute »@IBAction« und »@IBOutlet«

Sicher ist Ihnen aufgefallen, dass Xcode die Methode mit dem Attribut `@IBAction` und die Eigenschaft mit dem Attribut `@IBOutlet` gekennzeichnet hat. Der Interface Builder erkennt anhand dieser Attribute, zu welchen Elementen der Programmierer Verbindungen erlaubt bzw. herstellen darf. Der Interface Builder ist jener Teil von Xcode, der zur Gestaltung grafischer Benutzeroberflächen dient.

Endlich eigener Code

Sie können die App zwischenzeitlich noch einmal starten. Am Verhalten der App hat sich aber nichts verändert. Die Methode `hwButtonTouch` ist ja noch leer, und auch die zusätzliche Eigenschaft `textView` ist noch ungenutzt. Aber das ändert sich jetzt endlich! Jedes Mal, wenn der Button Hello World berührt wird, soll das Textfeld um eine Zeile mit Datum und Uhrzeit ergänzt werden. In einer ersten Testversion könnte der Code wie folgt aussehen:

```
// Projekt Hello iOS, Datei ViewController.swift
import UIKit
class ViewController: UIViewController {
  @IBOutlet weak var textView: UITextView!
```

```
  @IBAction func hwButtonTouch(_ sender: UIButton) {
    let now = Date()
    let formatter = DateFormatter()
    formatter.dateFormat = "d.M.yyyy H:mm:ss"
    textView.text? += formatter.string(from: now) + "\n"
  }
}
```

Auf den Abdruck der standardmäßig vorhandenen, aber leeren Methoden `viewDidLoad` und `didReceiveMemoryWarning` habe ich verzichtet. In diesem Beispiel können Sie diese Methoden aus dem Code löschen, wir brauchen sie nicht.

Outlets wie die Eigenschaft `textView` sind Optionals. Es liegt in der Natur von Optionals, dass diese den Zustand `nil` haben können. Tatsächlich ist dies aber nur vor der Initialisierung der Ansicht der Fall. Sobald die Ansicht auf einem iOS-Gerät sichtbar wird und Action-Methoden ausführen kann, können Sie sich darauf verlassen, dass Outlet-Eigenschaften nicht `nil` sind. Genau genommen gilt dies ab dem Zeitpunkt, zu dem im View-Controller die Methode `viewDidLoad` ausgeführt wurde (siehe Abschnitt 16.3, »Die UIViewController-Klasse«).

Hintergrundinformationen zum Umgang mit Datum und Uhrzeit können Sie bei Bedarf in Kapitel 9 nachlesen. Somit bleibt nur noch der Ausdruck `textView.text` zu erklären: Ein Blick in die Dokumentation des `UITextView`-Steuerelements zeigt, dass der Inhalt des Textfelds über die Eigenschaft `text` gelesen und verändert werden kann. Der Datentyp von `text` lautet `String?`, es handelt sich also um ein Optional, das beim Zugriff explizit durch ein nachgestelltes Fragezeichen oder Ausrufezeichen ausgepackt werden muss.

Der schnellste Weg zur Dokumentation

Wenn Sie zum ersten Mal ein `UITextView`-Steuerelement nutzen, kennen Sie die Elemente dieser Klasse noch nicht. Eine kurze Beschreibung der Klasse erhalten Sie, wenn Sie das Schlüsselwort `UITextView` im Code zusammen mit alt anklicken. Am unteren Rand der eingeblendeten Infobox befindet sich ein Link auf die Klassenreferenz, die in einem eigenen Fenster geöffnet wird (siehe Abbildung 15.6).

Jetzt ist es höchste Zeit, die App im Simulator endlich auszuprobieren. Einige Klicks auf den Button beweisen, dass das Programm wie erwartet funktioniert (siehe Abbildung 15.3). Einen Preis für innovatives Layout wird es freilich nicht gewinnen.

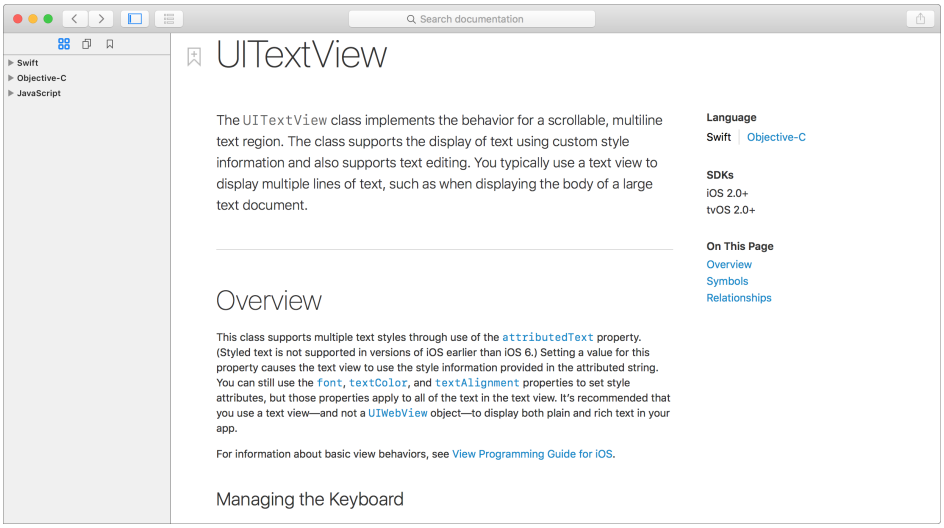

Abbildung 15.6 Der Hilfebrowser von Xcode

15.4 Actions und Outlets für Fortgeschrittene

Sie haben nun eine erste Vorstellung davon, wie Actions und Outlets funktionieren. Dieser Abschnitt weist auf einige Besonderheiten im Umgang mit Actions und Outlets hin.

Eine Action für mehrere Steuerelemente

Es ist zulässig, ein und dieselbe Action-Methode für mehrere Steuerelemente zu verwenden. Dazu richten Sie zuerst für ein Steuerelement die Methode ein. Danach führen Sie eine ctrl-Drag-Operation für das zweite Steuerelement aus, wobei Sie als Ziel die bereits vorhandene Methode verwenden. Achten Sie darauf, dass die gesamte Methode blau unterlegt wird – dann hat Xcode erkannt, dass Sie nicht eine Action *in* der Methode einfügen möchten (das funktioniert nicht), sondern dass Sie das Steuerelement mit der vorhandenen Methode verbinden möchten.

Naturgemäß müssen Sie die Methode nun ein wenig modifizieren: Erst mit einer Auswertung des `sender`-Parameters können Sie erkennen, welches Steuerelement den Aufruf der Methode ausgelöst hat. Wenn Sie beispielsweise mehrere Buttons mit einer Methode verbunden haben, können Sie wie folgt den Text des Buttons ausgeben:

```
@IBAction func btnAction(_ sender: UIButton) {
  print(sender.currentTitle!)
}
```

Ein Outlet für mehrere Steuerelemente (Outlet Collections)

Auch der umgekehrte Fall ist möglich – Sie können mehrere Steuerelemente über ein Outlet ansprechen. Genau genommen handelt es sich dann nicht mehr um ein einfaches Outlet, sondern um eine *Outlet Collection*. Dazu markieren Sie das erste Steuerelement, ziehen es mit ctrl-Drag in den Controller-Code und wählen Connection = Outlet Collection (siehe Abbildung 15.7). Xcode erzeugt anstelle einer einfachen Outlet-Variablen nun ein Array:

```
@IBOutlet var allButtons: [UIButton]!
```

In der Folge verbinden Sie auch die weiteren Steuerelemente durch ctrl-Drag mit diesem Array. Xcode ist leider nicht in der Lage, mehrere markierte Steuerelemente auf einmal zu verbinden. Im Code können Sie nun unkompliziert Schleifen über alle so verbundenen Steuerelemente bilden.

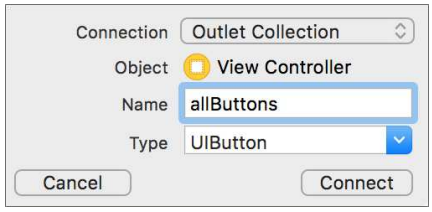

Abbildung 15.7 Dialog zum Einrichten einer Outlet-Collection

Actions oder Outlets umbenennen

Wenn Sie im Code Actions oder Outlets einfach umbenennen, funktioniert Ihre App nicht mehr. Xcode merkt sich die Verknüpfung zu Ihren Methoden bzw. Eigenschaften in Form von Zeichenketten. Nach der Umbenennung ist keine korrekte Zuordnung mehr möglich.

Abhilfe: Klicken Sie im Storyboard-Editor das betreffende Steuerelement mit der rechten Maus- oder Trackpad-Taste an. Ein Kontextmenü zeigt nun alle Zuordnungen zu Actions oder Outlets (siehe Abbildung 15.8). Dort löschen Sie die Action- oder Outlet-Verknüpfung, die Sie umbenannt haben. Sollten Sie das vergessen, wird die App abstürzen, wobei die Fehlermeldung *unrecognized selector* lautet.

Anschließend wiederholen Sie die ctrl-Drag-Operation und ziehen die Verknüpfungslinie direkt zur schon vorhandenen, von Ihnen umbenannten Methode oder Eigenschaft. Damit wird die Verknüpfung wieder neu hergestellt, und Sie müssen nicht alle Einstellungen wiederholen.

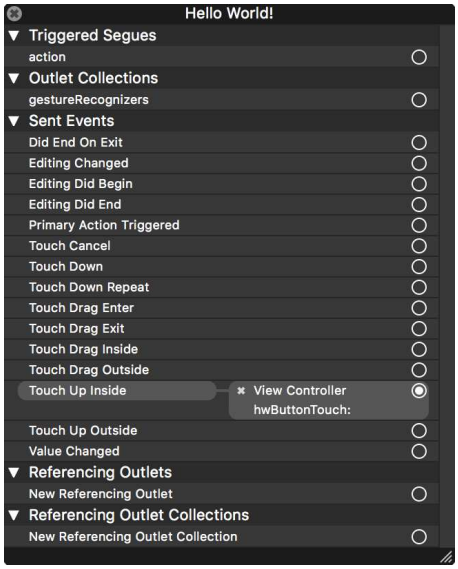

Abbildung 15.8 Die rechte Maus- oder Trackpad-Taste führt in eine Liste aller Actions und Outlets eines Steuerelements.

Steuerelemente kopieren

Wenn Sie Steuerelemente mit ⌘+C und ⌘+V kopieren, werden dabei alle möglichen unsichtbaren Attribute und Eigenschaften mitkopiert, unter anderem zugeordnete Actions. Oft erspart Ihnen das eine wiederholte Einstellung dieser Merkmale, aber mitunter führt dieses Verhalten zu unerwarteten Nebenwirkungen. Ein Klick auf das Steuerelement mit der rechten Maus- oder Trackpad-Taste offenbart alle zugeordneten Outlets und Actions.

15.5 Layout optimieren

Es ist Ihnen sicher aufgefallen, dass das Layout unserer App, also die Anordnung und Größe der Steuerelemente, verbesserungswürdig ist. Die Größe der Steuerelemente ist willkürlich. Wenn Sie die App in verschiedenen iOS-Geräten ausprobieren, werden Sie feststellen, dass teilweise große Teile des Bildschirms ungenutzt bleiben, während das Textfeld bei anderen Geräten womöglich sogar abgeschnitten und unvollständig dargestellt wird. Besonders deutlich werden die Layoutdefizite, wenn Sie den iOS-Simulator mit HARDWARE • ROTATE in das Querformat drehen.

Wie würden unsere Layoutwünsche denn aussehen?

- Der Button soll ohne unnötige Abstände links oben im Bildschirm dargestellt werden.
- Das Textfeld soll darunter platziert sein.
- Es soll die gesamte verbleibende Größe des Bildschirms nutzen.
- Es soll seine Größe bei einer Drehung des Geräts automatisch anpassen.

Momentan wird unser Programm diesen Wünschen deswegen nicht gerecht, weil wir die Position und Größe der Steuerelemente absolut festgelegt haben. Wir haben die Steuerelemente im View weitgehend nach Gutdünken platziert.

Layoutregeln

Die Lösung, die Xcode bzw. eigentlich das UIKit, also das Framework zur iOS-Programmierung, hierfür anbietet, heißt Layoutregeln (*Constraints*). Sie können also für jedes Steuerelement Regeln aufstellen, die dieses einhalten soll. Das UIKit bemüht sich dann, in Abhängigkeit von der gerade vorliegenden Form und Größe des iOS-Geräts, allen Regeln gerecht zu werden. Beispiele für derartige Regeln sind:

- Der horizontale Abstand zwischen dem Steuerelement A und seinem nächstgelegenen linken oder rechten Nachbarn soll 8 Punkte betragen. Bei Retina-Geräten mit doppelter Auflösung entspricht das 16 Pixeln.
- Der vertikale obere Abstand zwischen dem Steuerelement A und dem Bildschirmrand soll 16 Punkt betragen.
- Steuerelement A soll genauso breit sein wie Steuerelement B.
- Steuerelement A soll innerhalb seines Containers vertikal und/oder horizontal zentriert werden.
- Die linken Ränder der Steuerelemente A, B und C sollen in einer Linie verlaufen.

Wenn wir also erreichen möchten, dass die Steuerelemente der Hello-World-App wie oben formuliert angeordnet werden, müssen wir nur die entsprechenden Regeln formulieren. Mit etwas Erfahrung gelingt dies rasch; gerade Einsteiger in die iOS-Programmierung scheitern aber oft an der damit verbundenen Komplexität.

Glücklicherweise macht Xcode 8 es einfacher denn je, Layoutregeln sofort zu testen, ohne die App jedes Mal neu starten zu müssen: Dazu verändern Sie einfach in der Fußleiste von Xcode das iOS-Device und seine Orientierung. Xcode passt dann die View an die neue Gerätegröße an, und Sie erkennen sofort, ob die Steuerelemente wunschgemäß platziert werden.

Layoutregeln für den »Hello-World«-Button

Eine detaillierte Erklärung der Layoutregeln erhalten Sie in Abschnitt 16.5. An dieser Stelle möchte ich Ihnen nur rezeptartig erklären, wie Sie das Layout der Hello-World-App korrekt einstellen. Dazu klicken Sie im Storyboard zuerst auf den Button HELLO WORLD, dann auf den Button PIN, der sich rechts unten im Editor befindet (siehe Abbildung 15.9).

Abbildung 15.9 Die vier winzigen Layout-Button befinden sich rechts unten im Storyboard-Editor.

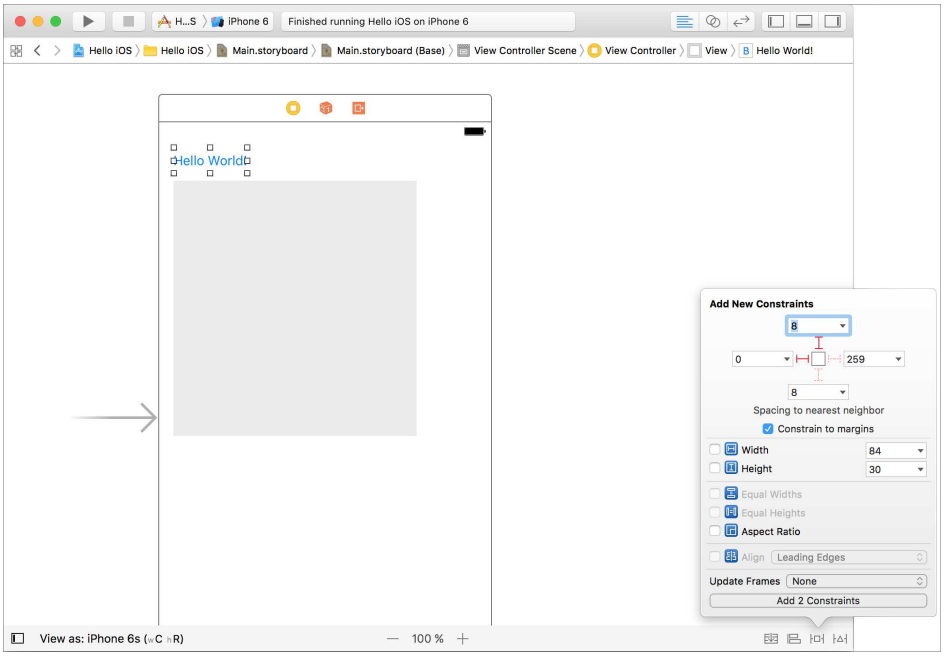

Abbildung 15.10 Layoutregeln für den Hello-World-Button festlegen

Damit erscheint der Dialog ADD NEW CONSTRAINTS zur Einstellung diverser Abstände (siehe Abbildung 15.10). Dort klicken Sie zuerst die Verbindungsstege für die Abstände nach oben bzw. zur linken Seite an, sodass diese Stege durchgängig rot angezeigt werden. Anschließend geben Sie für den Abstand nach oben 8 Punkte und für den seit-

lichen Abstand 0 Punkte an. Standardmäßig ist die Option Constrain to margins aktiv. Sie bewirkt, dass diese Abstände relativ zu einem vom jeweiligen iOS-Gerät vorgegebenen Standardrahmen gerechnet werden. Zuletzt schließen Sie den Dialog mit dem Button Add 2 Constraints. Damit werden zwei neue Regeln zur Positionierung des Buttons festgelegt.

Überraschenderweise führen die neuen Regeln nicht zu einer Veränderung der Position des Steuerelements. Vielmehr zeigen orange strichlierte Linien an, wo der Button bei der Programmausführung platziert wird (siehe Abbildung 15.11), wenn die Position zur Laufzeit eine andere ist als momentan im Storyboard-Editor. Außerdem visualisieren zwei orange Stege die von Ihnen aufgestellten Regeln. Aus den Zahlenwerten geht hervor, um wie viele Punkte das Element aktuell falsch positioniert ist.

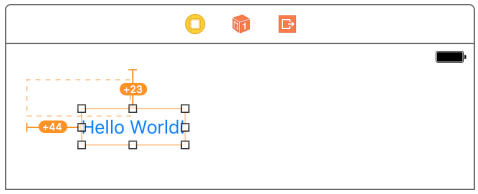

Abbildung 15.11 Der Storyboard-Editor zeigt, wo der Button später platziert wird.

Um den Button den neuen Regeln entsprechend zu positionieren, klicken Sie auf den Button Resolve Auto Layout Issues und führen dort das Kommando Selected Views • Update Frames aus.

Stack-View-Button rückgängig machen

Die Buttons Align, Pin und Resolve Auto Layout Issues führen jeweils in Dialoge oder Menüs, in denen Sie die jeweilige Aktion auswählen bzw. bestätigen. Es besteht also keine Gefahr, dass Sie irrtümlich eine Aktion durchführen, die Sie nicht wollen.

Ganz anders verhält sich der Button Stack View: Sofern im Storyboard-Editor gerade ein Steuerelement markiert ist bzw. mehrere Steuerelemente markiert sind, werden diese in eine Stack-View verpackt, also in einen Container, der bei der Anordnung der Steuerelemente helfen soll. Diese Aktion erfolgt sofort, ohne Rückfrage und ohne visuelles Feedback, also oft ungewollt und unbemerkt.

Ungewollte Stack-View-Verschachtelungen erkennen Sie, wenn Sie im Storyboard-Editor die Seitenleiste Document Outline aktivieren. Sie zeigt die hierarchische Struktur aller Steuerelemente im Storyboard-Editor. Dort können Sie bei Bedarf die betroffenen Steuerelemente aus der Stack-View herausziehen (siehe Abbildung 15.12). Danach können Sie die Stack-Views in der Seitenleiste anklicken und löschen.

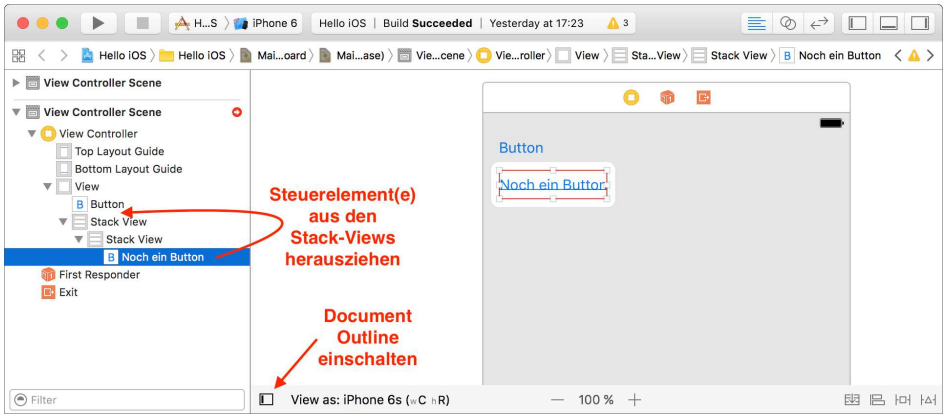

Abbildung 15.12 Ungewollte Stack-View-Verschachtelungen auflösen

Stack-Views richtig einsetzen

Stack-Views sind ein durchaus nützliches Feature, das dabei hilft, allzu komplizierte Layout-Regeln zu vermeiden. Den richtigen Umgang mit Stack-Views erläutere ich Ihnen in Abschnitt 16.6.

Die Document-Outline-Seitenleiste verstehen

Die Document-Outline zeigt, wie die Steuerelemente (Views) innerhalb eines View-Controllers ineinander verschachtelt sind (siehe Abbildung 15.12), und sie hilft dabei, das gewünschte Element auszuwählen, wenn zwei Elemente übereinander platziert sind. Innerhalb des Objektbaums erhält jedes Element standardmäßig einen Namen, der seiner Klasse entspricht – z. B. `View` für ein `UIView`-Objekt. Bei einigen Steuerelementtypen, z. B. bei Buttons und Labels, wird der dort eingetragene Text angezeigt – also `Noch ein Button` statt einfach `Button`.

Sie können im Objektbaum alle Objekte durch einfaches Anklicken umbenennen. Das ist dann sinnvoll, wenn es mehrere gleichartige Objekte gibt, die Sie im Storyboard-Editor leichter identifizieren wollen. Beachten Sie aber, dass die Objektnamen in der Document-Outline-Seitenleiste ausschließlich für den Storyboard-Editor von Xcode gelten!

Im Code Ihrer View-Controller-Klasse haben Sie keine Möglichkeit, den Namen festzustellen, den Sie einem Objekt in der Outline-Seitenleiste gegeben haben. Sie können zur Identifizierung von Steuerelementen im Code auf die `tag`-Eigenschaft zurückgreifen. In dieser auch in Xcode zugänglichen Eigenschaft können Sie allerdings nur eine ganze Zahl speichern. Eine zweite Möglichkeit besteht darin, eine eigene Klasse mit einer neuen Eigenschaft zu definieren. Ein Beispiel dafür finden Sie in Abschnitt 20.3, »Focus Engine«.

Layoutregeln für das Textfeld

Nun ist das Textfeld an der Reihe: Nachdem Sie dieses angeklickt haben, öffnen Sie wieder mit dem PIN-Button den Dialog ADD NEW CONSTRAINTS. Dort stellen Sie die folgenden Abstände ein (siehe Abbildung 15.13):

- Links: 0 Punkte. Dieser Abstand gilt wegen der Option CONSTRAIN TO MARGINS relativ zum Standardrahmen.
- Oben: 0 Punkte. Dieser Abstand wird relativ zum nächstgelegenen Steuerelement gerechnet, in diesem Fall also zum Button.
- Rechts: 0 Punkte. Dieser Abstand gilt wieder relativ zum Standardrahmen.
- Unten: 16 Punkte. Auch dieser Abstand gilt relativ zum Standardrahmen. Dieser sieht nach unten aber keinen Rand vor. Damit das Textfeld von allen Rändern gleich weit entfernt ist, muss hier ein etwas größerer Wert angegeben werden.

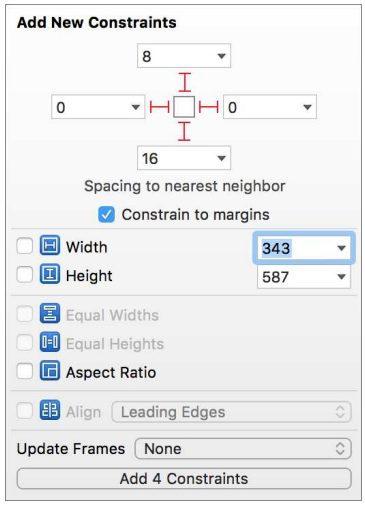

Abbildung 15.13 Layoutregeln für das Textfeld

ADD 4 CONSTRAINTS beendet die Eingabe und fügt vier Regeln hinzu. Um auch das Textfeld gemäß der neuen Regeln korrekt zu platzieren, klicken Sie nochmals auf den Button RESOLVE AUTO LAYOUT ISSUES und führen dort das Kommando SELECTED VIEWS • UPDATE FRAMES aus.

Damit sollten nun beide Steuerelemente korrekt angeordnet sein. Ändern Sie nun in der Fußleiste des Storyboard-Editors das iOS-Device und die Ausrichtung. Der Ort und die Größe der Steuerelemente sollten nun immer korrekt an die jeweilige Bildschirmgröße angepasst werden (siehe Abbildung 15.14).

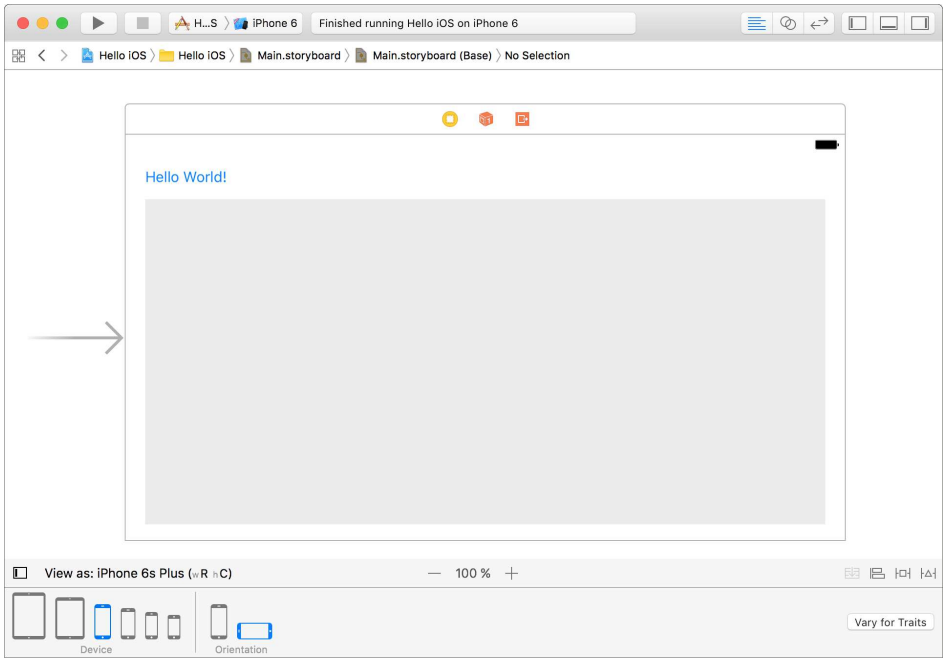

Abbildung 15.14 Die Hello-World-App im iPhone-7-Plus-Querformat

Wenn es Probleme gibt

Der Umgang mit Layoutregeln ist schwierig und führt häufig dazu, dass sich Xcode über fehlende oder über zueinander im Konflikt stehende Regeln beklagt. In Abschnitt 16.5, »Auto Layout«, folgt eine Menge weiterer Details zu diesem Thema. Bis dahin vertröste ich Sie hier mit einigen Tipps:

- Sobald Sie *eine* Regel für ein Steuerelement festlegen, müssen Sie die Größe und Position des Steuerelements *vollständig* durch Regeln bestimmen. Mit anderen Worten: Solange es gar keine Regeln gibt, betrachtet Xcode das Steuerelement als unbestimmt und meckert nicht. Sobald Sie aber beginnen, Regeln festzulegen, müssen Sie dies so tun, dass keine Unklarheiten verbleiben.

  Die Anzahl der erforderlichen Regeln ist nicht bei jedem Steuerelement gleich. Manche Steuerelemente können ihre optimale Größe aus dem Inhalt selbst ermitteln. Das trifft z. B. bei einem Button zu. Hier reichen also Regeln, die die Position festlegen. Bei anderen Steuerelementen müssen Sie die Größe selbst einstellen – und das erfordert zwei weitere Regeln.

- Sie können vorhandene Regeln nicht ohne Weiteres ändern. Die Dialoge Add New Alignment Constraints bzw. Add New Constraints ersetzen bzw. verändern nicht vorhandene Regeln, sondern definieren zusätzliche Regeln. Das führt oft zu Regeln, die sich widersprechen. Einen Überblick über alle Regeln erhalten Sie, wenn

Sie die Seitenleiste des Storyboard-Editors einblenden (Editor • Show Document Outline). Dort finden Sie eine Liste aller Constraints. Wenn Sie eine der Regeln anklicken, wird die betreffende Regel markiert.

- Bei kleinen Projekten ist es bei Problemen oft am einfachsten, alle Regeln zu löschen und noch einmal von vorne zu beginnen. Dazu klicken Sie auf den Button Resolve Auto Layout Issues und führen All Views in View Controller • Clear Constraints aus.

15.6 Textgröße mit einem Slider einstellen

Als letzte Erweiterung für das Programm fügen wir diesem nun neben dem Button noch einen Slider hinzu, mit dem die Schriftgröße des Textfelds verändert werden kann (siehe Abbildung 15.15).

Abbildung 15.15 Die Hello-World-App mit einem Slider zur Einstellung der Textgröße

Das Slider-Steuerelement hinzufügen

Sie finden das Steuerelement in der Xcode-Objektbibliothek unter dem Namen *Slider*. Zur Positionierung stellen Sie zuerst mit dem Pin-Button die horizontalen Abstände ein: Der linke Abstand zum Hello World-Button soll 24 Punkte betragen, der rechte

Abstand zum Rand 0 Punkte. Definieren Sie aber keine Regeln für die vertikale Position!

In einem zweiten Schritt markieren Sie nun mit ⇧ sowohl den Hello World-Button als auch den Slider. Mit dem Align-Button öffnen Sie dann den Dialog New Add Align Constraints und wählen dort die Option Vertical Centers. Diese Regel bewirkt, dass der Button und der Slider vertikal mittig angeordnet werden, was in diesem Fall harmonisch aussieht. Über den Button Resolve Auto Layout Issues führen Sie nun das Kommando Selected Views • Update Frames aus, damit der Slider im Storyboard-Editor an der richtigen Stelle angezeigt wird.

Mit dem Slider soll die Textgröße in einem Bereich zwischen 12 und 30 Punkt verändert werden. Dazu wählen Sie den Slider aus und stellen im Attributinspektor die folgenden Eigenschaften ein:

- Minimum: 12
- Maximum: 30
- Current: 16

Den Slider mit einer Methode verbinden

Damit sind die Arbeiten an der Oberfläche abgeschlossen, und wir können uns wieder der Programmierung zuwenden: Öffnen Sie den Assistenz-Editor, und verschieben Sie den Slider mit ctrl in den Code-Bereich. Die Verbindungsparameter stellen Sie wie folgt ein:

- Connection = Action: Wir wollen in einer Methode auf das Verschieben des Sliders reagieren.
- Name: Die Methode muss einen Namen bekommen. Ich habe mich für `sliderMove` entschieden.
- Type = UISlider: In der Methode müssen wir die aktuelle Position des Sliders herausfinden. Deswegen soll die Instanz des Sliders an die Methode übergeben werden.
- Event = Value Changed: Die Defaulteinstellung passt hier gut – andere Ereignisse interessieren uns nicht.
- Arguments = Sender: Damit wird die Instanz des Sliders als Parameter an die Methode übergeben. Den Datentyp des Parameters haben wir ja bereits mit `UISlider` festgelegt.

Zu Testzwecken bauen wir in die Methode vorerst nur die `print`-Funktion ein, um eine Veränderung des Sliders in Xcode verfolgen zu können. Uns interessiert die `value`-Eigenschaft, die die Slider-Position im eingestellten Wertebereich als Fließkommazahl liefert.

```
// wird bei jeder Slider-Bewegung ausgeführt
@IBAction func sliderMove(_ sender: UISlider) {
  // Testausgabe im Debug-Fenster
  print(sender.value)
}
```

Jetzt geht es nur noch darum, die Schrift des Textfelds entsprechend zu verändern. Dazu lesen wir mit `textView.font?` die aktuelle Font-Instanz aus, bilden daraus mit der Methode `withSize` eine neue Instanz in der gewünschten Größe und weisen diese der `font`-Eigenschaft des Textfelds wieder zu. Da `fontWithSize` einen `CGFloat`-Parameter erwartet, muss die Fließkommazahl von `sender.value` in den `CGFloat`-Typ umgewandelt werden. `CGFloat` ist auf 32-Bit-Architekturen ein `Float`, auf 64-Bit-Architekturen aber ein `Double`.

```
@IBAction func sliderMove(_ sender: UISlider) {
  textView.font =
    textView.font?.withSize(CGFloat(sender.value))
}
```

15.7 Apps auf dem eigenen iPhone/iPad ausführen

Den iOS-Simulator in Ehren, aber natürlich wollen Sie Ihre Programme auch auf »richtiger« Hardware testen. Während dazu früher ein kostenpflichtiger Apple-Developer-Account erforderlich war, ist das Ausführen eigener Apps auf eigenen Geräten seit Mitte 2015 kostenlos möglich (»Free Provisioning«). Dazu verbinden Sie im Dialog PREFERENCES • ACCOUNTS Xcode mit Ihrer Apple ID. Außerdem muss Ihr iOS-Gerät durch ein USB-Kabel mit dem Computer verbunden sein. Nach diesen Vorbereitungsarbeiten können Sie das iOS-Gerät in der Symbolleiste von Xcode auswählen.

Wenn Sie ⌘+R drücken bzw. den RUN-Button anklicken, überträgt Xcode die Hello-World-App auf das angeschlossene iPhone oder iPad und startet sie dort. Das gelingt nur, wenn Ihr Smartphone oder Tablet entsperrt ist. Wenn also eine Ziffern- oder Fingerabdruck-Sperre aktiv ist, müssen Sie das Gerät zuerst einschalten, bevor Sie Ihre App in Xcode starten. Bemerkenswert ist, dass trotz der externen Programmausführung die Debugging-Funktionen von Xcode aktiv bleiben. Wenn Sie also z. B. einen Breakpoint setzen, wird die App an dieser Stelle angehalten. Sie können in Xcode den Zustand der Variablen ergründen und das Programm dann wieder fortsetzen.

Die App bleibt jetzt auf dem iPhone oder iPad. Sie kann dort losgelöst von Xcode ausgeführt werden – dann aber ohne Debugging-Möglichkeiten. Sie können Ihre App wie jede andere installierte App problemlos wieder löschen, indem Sie sie zuerst länger anklicken und dann auf das x-Symbol drücken.

Einschränkungen des Free Provisioning

Apple bezeichnet das Verfahren zum Ausführen von Apps auf iOS-Geräten ohne Apple-Developer-Account als *Free Provisioning*. Dabei gibt es aber Einschränkungen: Der Test von einigen Zusatzfunktionen erfordert weiterhin einen kostenpflichtigen Apple-Developer-Account. Das gilt z. B. für In-App-Käufe, den Datenaustausch mit der iCloud oder die Verwendung der Apple-Pay-Funktionen.

Apple Developer Program

Bevor Sie eine App in den App Store hochladen können, müssen Sie Ihre App speziell vorbereiten und signieren (siehe Kapitel 29, »App Store und Co.«). Das ist nur mit dem Schlüsselsystem des Apple Developer Program möglich. Die Mitgliedschaft hat auch andere Vorteile – etwa den unkomplizierten Zugang zu Beta-Versionen von iOS, macOS und Xcode, den Zugang zu Entwicklerforen etc.:

https://developer.apple.com/programs

Dieses Service-Paket lässt sich Apple mit zurzeit 100 EUR pro Jahr bezahlen. Im Gegensatz zu früher, als es verschiedene Entwicklerprogramme für iOS, macOS und Safari gab, hat Apple diese Programme nun zu einem einzigen verbunden.

Eine vorhandene oder eine neue Apple-ID verwenden?

Bevor Sie sich dem Entwicklerprogramm anschließen, müssen Sie sich überlegen, welche Apple-ID Sie hierfür verwenden. Normalerweise spricht nichts gegen Ihre gewöhnliche Apple-ID. Sollten Sie diese ID aber schon im Rahmen von *iTunes Connect* zum Verkauf von Musik oder Büchern nutzen, dann benötigen Sie eine zweite Apple-ID für das Entwicklerprogramm.

Sie können dem Entwicklerprogramm wahlweise als Einzelperson oder als Team beitreten. Als Einzelperson benötigen Sie dazu lediglich eine Kreditkarte. Nach Abschluss des Bezahlprozesses kann es ein paar Minuten dauern, bis Ihr Entwicklerzugang freigeschaltet wird und Sie die entsprechende *Welcome*-E-Mail erhalten.

Nun können Sie in den Xcode-Einstellungen im Dialogblatt Accounts mit Add Apple ID Ihre Apple-ID mit Xcode verbinden. Von dort gelangen Sie mit View Details in einen weiteren Dialog, in dem Sie Schlüssel generieren können (siehe Abbildung 15.16). Vorerst benötigen Sie lediglich einen Schlüssel zur iOS-Entwicklung (also den Eintrag iOS Development).

Nach diesen Vorbereitungsarbeiten können Sie nun ein mit einem USB-Kabel angeschlossenes iOS-Gerät in der Symbolleiste von Xcode auswählen. Das Gerät wird damit in das Entwicklungsprogramm aufgenommen. Insgesamt dürfen Sie pro Jahr

maximal 500 Geräte mit Ihrem Konto verbinden: 100 iPhones, 100 iPods, 100 iPads sowie je 100 Apple-TV- und Apple-Watch-Geräte. Einen Überblick über alle mit Ihrem Konto verbundenen iOS-Geräte finden Sie auf der Webseite des Entwicklerprogramms:

https://developer.apple.com/account/ios/device

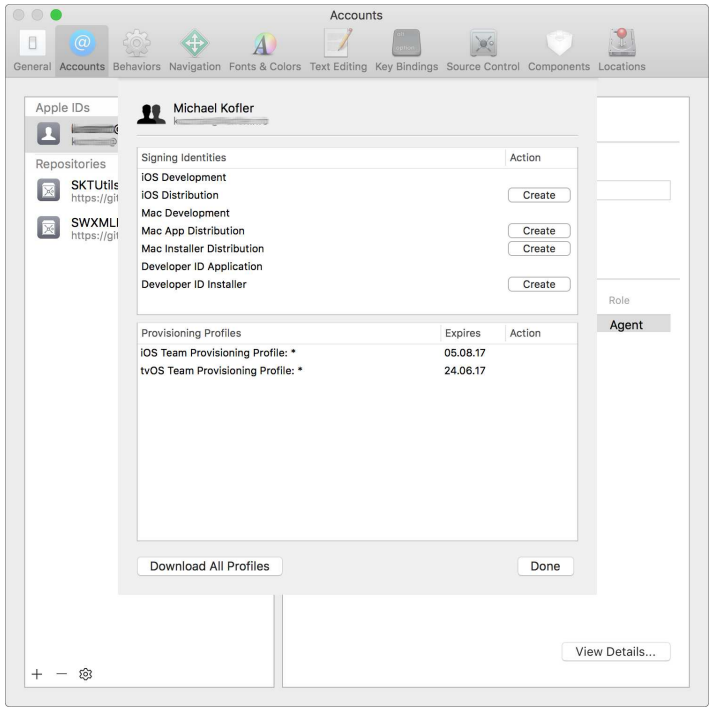

Abbildung 15.16 Verwaltung der Schlüssel des iOS-Entwicklerprogramms in Xcode

15.8 Komponenten und Dateien eines Xcode-Projekts

Wenn Sie in Xcode ein neues iOS-Projekt starten, besteht dieses standardmäßig schon aus einer Menge Dateien (siehe Abbildung 15.17) – und dabei bleibt es nicht. Dieser Abschnitt gibt Ihnen einen kurzen Überblick darüber, welche Datei welchen Zweck hat. Detaillierte Erläuterungen zu vielen Dateien folgen dann in den weiteren Kapiteln.

- `AppDelegate.swift` enthält Code zur Verarbeitung von Ereignissen des App-Lebenszyklus (siehe Abschnitt 16.4, »Phasen einer iOS-App«).

- `LaunchScreen.xib` enthält eine spezielle Ansicht der App, die während des Starts als eine Art Willkommensdialog angezeigt wird (siehe Abschnitt 29.1, »iOS-Artwork (Icons, Launch Screen)«).
- `Assets.xcassets` dient als Container für die Bilddateien der App. Dazu zählen neben dem Icon der App auch alle anderen Bitmaps, die Sie irgendwann anzeigen möchten. Die Besonderheit von Xcassets-Dateien besteht darin, dass Bitmaps in mehreren Auflösungen gespeichert werden können. Bei der Ausführung verwendet iOS dann automatisch die Datei, die am besten zum Display des iOS-Geräts passt (siehe Abschnitt 16.8, »Image-Views und Xcassets« und in Abschnitt 29.1, »iOS-Artwork (Icons, Launch Screen)«).
- `Info.plist` enthält diverse Projekteinstellungen in Form einer sogenannten Property List (Key-Value-Datei).
- `Main.storyboard` beschreibt das Aussehen und die Eigenschaften der Ansichten (View-Controller) einer App.
- `ViewController.swift` enthält den Controller-Code der ersten Ansicht des Storyboards. Für jede weitere Ansicht im Storyboard müssen Sie in der Regel eine weitere Swift-Datei hinzufügen, die eine von `UIViewController` abgeleitete Klasse definiert (siehe Kapitel 17, »iOS-Apps mit mehreren Ansichten«).

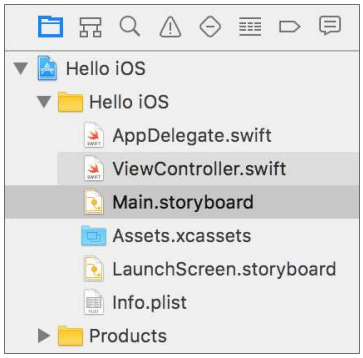

Abbildung 15.17 Überblick über die Code-Dateien im Projektnavigator von Xcode

Dateien im Navigator verschieben

Sie können die Dateien des Projekts im Navigator verschieben und in Gruppen gliedern. Zwei Dinge sind in diesem Zusammenhang bemerkenswert: Zum einen spielt es für Xcode keine Rolle, in welcher Gruppe die Dateien sich befinden. Xcode findet in jedem Fall alle zum Kompilieren erforderlichen Dateien. Und zum anderen sind Gruppen *keine* Unterverzeichnisse im Projektverzeichnis. Gruppen helfen bei der Organisation der Dateien, haben aber keinen Einfluss darauf, wo Dateien tatsächlich gespeichert werden. Der Projektnavigator ist also kein Abbild des Dateisystems!

Weitere Dateien

Bei »richtigen« Apps, die also nicht nur Test- oder Beispielcharakter haben, kommen zu den anfänglich vorhandenen Dateien zumeist viele Dateien hinzu:

- Weitere Swift-Code-Dateien verbinden Objekte Ihrer Oberfläche mit eigenen Klassen bzw. bilden die innere Logik Ihres Programms ab, also das Datenmodell gemäß des MVC-Musters (siehe Abschnitt 16.1, »Model-View-Controller (MVC)«).
- Zusätzliche Lokalisierungsdateien enthalten Zeichenketten für alle Sprachen, in denen die App später ausgeführt werden kann (siehe Abschnitt 29.4, »Mehrsprachige Apps«).
- Ja nach Zielsetzung der App sind außerdem Text-, XML-, HTML-, Datenbank- sowie Audio- und Video-Dateien erforderlich. Diese Dateien werden zusammen mit der App ausgeliefert (»Bundle-Dateien«).

Test- und Produktgruppe

Neben der eigentlichen Projektgruppe, deren Name mit dem Projektnamen übereinstimmt, kann ein Projekt bis zu drei weitere Gruppen aufweisen:

- `projektnameTests` und `projektnameUITests` enthält Code und Einstellungen zum automatisierten Test Ihres Projekts. Das zugrunde liegende XCTest-Framework hat eine ähnliche Zielsetzung wie Unit Tests in anderen Programmiersprachen. In diesem Buch gehe ich darauf allerdings nicht ein.
- `Products` enthält das kompilierte Programm. Bei der iOS-App-Entwicklung werden die hier enthaltenen Dateien aber selten benötigt, weil die Ausführung von Apps durch Xcode automatisiert ist und eine Weitergabe von Apps an andere Benutzer nur über den App Store möglich ist.

Kapitel 37
Breakout

Breakout ist ein klassisches Spielhallenspiel. Es geht darum, mit einem Ball rechteckige Steine (Bricks) abzuschießen. Es ist Aufgabe des Spielers, den Ball mit einem Schläger (Paddle) in dem nach unten offenen Spielfeld zu halten. Die erste Version des Spiels wurde 1976 von Atari produziert, wobei die Spiellogik nicht durch ein Programm, sondern in Hardware realisiert wurde.

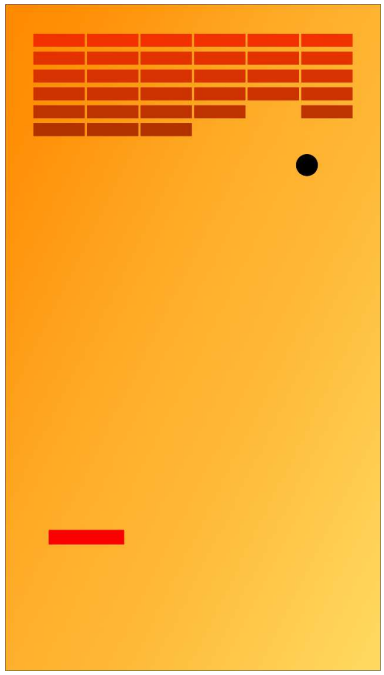

Abbildung 37.1 Das Spiel »Breakout«

Breakout hat übrigens schon seit Langem mit Apple zu tun: Steve Jobs, der damals für Atari arbeitete, überließ es Steve Wozniak, einen Prototyp für das Spiel zu bauen, bezahlte ihn dafür aber nur schäbig. Nichtsdestotrotz gründeten die beiden wenig später Apple.

https://de.wikipedia.org/wiki/Breakout_(Computerspiel)

Aus Entwicklersicht ist Breakout ein dankbares Beispiel für den Einstieg in die Spieleprogrammierung (siehe auch Kapitel 27, »SpriteKit«): Die hier präsentierte Version umfasst nur rund 250 Zeilen, wenn man den Begleit-Code wegrechnet (`CGOperators.swift`, `GameViewController.swift` etc.).

37.1 Programmaufbau

Die App wurde aus dem Standard-Template für iOS-SpriteKit-Spiele entwickelt. Die wichtigsten Dateien sind:

- `GameScene.sks` enthält die farbige Hintergrund-Bitmap für die einzige Spielszene.
- `GameScene.swift` enthält den Großteil des Codes.
- `GameViewController.swift` lädt und skaliert die GameScene-Datei.
- `CGOperators.swift` enthält Methoden und Operatoren zum eleganteren Umgang mit `CGXxx`-Strukturen (siehe Abschnitt 7.3, »CGFloat, CGPoint, CGSize und Co.«).

Das Spiel kann nur im Portrait-Modus gespielt werden. Andere Geräteausrichtungen wurden für das iPhone in den Projekteinstellungen deaktiviert bzw. für das iPad in `Info.plist` gelöscht (Key *Supported interface orientations (iPad)*).

In Abbildung 37.1 ist Ihnen vielleicht aufgefallen, dass unterhalb des Schlägers relativ viel freier Platz ist. Beim Test des Programms hat es sich als praktisch erwiesen, diesen Platz für die Spielsteuerung frei zu lassen. Das ermöglicht die Steuerung des Schlägers mit einem Finger, ohne dass der Finger den Schläger verdeckt.

SpriteKit-Funktionen

Mit Ausnahme der Hintergrund-Bitmap verwendet die App keine Sprites, sondern nur Shapes (also `SKShapeNode`-Objekte). Die App nutzt die Physik-Engine: einerseits zur Berechnung der Flugbahn des Balls, andererseits zur Erkennung von Kollisionen mit Steinen. Diese werden dann aus dem Spielfeld entfernt. Es gibt keine Aktionen (`SKAction`-Animationen).

Erweiterungsideen

In dem Spiel gibt es nur einen Level und nur ein Leben. Wenn der Ball aus dem Spielfeld fliegt, wird das Spiel bei der nächsten Berührung des Bildschirms neu gestartet. Wenn Sie Spaß an der App haben, können Sie diverse Erweiterungen programmieren:

- Counter mit Highscore-Speicherung
- Verwaltung mehrerer Leben
- Gestaltung weiterer Level, wobei die Steine abwechslungsreicher angeordnet werden können (z. B. pyramidenförmig)

- Level mit einem zweiten Ball, der zwischen Steinen eingeschlossen ist und erst aktiv wird, wenn er »befreit« wird
- nachrückende Steine (d. h., alle 30 Sekunden kommt oben eine zusätzliche Steinreihe hinzu; noch vorhandene Steine werden nach unten verschoben)
- zunehmende Ballgeschwindigkeit im Spielverlauf

37.2 Initialisierung

In der Methode `viewDidLoad` des View-Controllers wird die Spielszene geladen. Die Breite der Szene wird aus `GameScene.sks` übernommen (640 Punkt). Die Höhe wird so angepasst, dass sie den Proportionen des iOS-Geräts entspricht.

```
// Projekt ios-breakout
// Datei GameViewController.swift
override func viewDidLoad() {
  super.viewDidLoad()

  // Initialisierung des Zufallszahlengenerators
  srand48(Int(arc4random_uniform(100000000)))

  // Spielszene laden
  if let scene = GameScene(fileNamed:"GameScene") {
    let skView = self.view as! SKView
    scene.scaleMode = .aspectFill
    scene.size = CGSize(width: scene.size.width,
                        height: scene.size.width /
                                skView.bounds.width *
                                skView.bounds.height)
    skView.presentScene(scene)
  }
}
```

Control Center verbergen

Unter iOS können Sie durch das Hinaufstreichen am unteren Bildschirmrand das Control Center öffnen. Ganz lässt sich das nicht verhindern – aber wenn die Eigenschaft `prefersStatusBarHidden` den Zustand `true` zurückgibt, dann wird ein versehentliches Öffnen des Control Centers zumindest wesentlich unwahrscheinlicher.

```
override var prefersStatusBarHidden: Bool {
  return true
}
```

Statusvariablen

GameScene.swift beginnt mit der Deklaration einiger Variablen, die die Eckdaten des Spiels bzw. Spielfelds zusammenfassen:

```
// Projekt ios-breakout
// Datei GameScene.swift

class GameScene: SKScene {
// Objekte
  var paddle: SKShapeNode!
  var ball:   SKShapeNode!
  var label = SKLabelNode(fontNamed: "AvenirNext-Bold")

  var w:CGFloat = 0  // enthält Breite und Höhe des SKScene-
  var h:CGFloat = 0  // Objekts, wird in didMove eingestellt

  // Spielstatus
  var brickcounter = 0
  var status = GameStatus.waitForStart
}
```

Der aktuelle Spielstatus wird in status gespeichert. Diese Variable kann die Werte der folgenden Enumeration annehmen:

```
enum GameStatus {
  case waitForStart, running, won, lost
}
```

Für die Kollisionserkennung sind am Ende von GameScene.swift außerdem in der Struktur PhysCategory einige UInt32-Konstanten definiert:

```
struct PhysCategory {
  static let none:UInt32   = 0
  static let ball:UInt32   = 1
  static let paddle:UInt32 = 2
  static let brick:UInt32  = 4
  static let frame:UInt32  = 8
}
```

Der gesamte weitere Code befindet sich innerhalb der Klasse GameScene.

Initialisierung der Spielszene

didMove wird zum Einrichten der Spielszene ausgeführt. Der Code sollte ohne weitere Erklärungen verständlich sein:

```
override func didMove(to view: SKView) {
  w = self.frame.width
  h = self.frame.height

  // keine Gravitation
  self.physicsWorld.gravity = CGVector.zero

  // zur Erkennung, wann ein Ball einen Stein trifft
  self.physicsWorld.contactDelegate = self

  // Spielobjekte zusammenstellen
  setupBricks(rows: 6, cols: 6)
  setupPaddle()
  setupBall()
  setupText()  // für 'Game Over'

  // Ball in Spielraum einsperren
  self.physicsBody = SKPhysicsBody(edgeLoopFrom: self.frame)
  with(self.physicsBody!) {
    $0.categoryBitMask    = PhysCategory.frame
    $0.collisionBitMask   = PhysCategory.ball
    $0.contactTestBitMask = PhysCategory.none
  }
}
```

Um bei der Einstellung der Eigenschaften des physikalischen Körpers die wiederholte Nennung von `brick.physicsBody!` zu vermeiden, habe ich die `with`-Funktion verwendet, die schon in Abschnitt 27.7, »Kollisionserkennung«, vorkam. Sie ist am Ende von `GameScene.swift` deklariert:

```
func with<T>(_ object: T, closure: (T)->()) {
  closure(object)
}
```

Spielsteine einrichten

`setupBricks` richtet `rows` Zeilen mit jeweils `cols` Steinen ein. In `brickcounter` wird die Gesamtanzahl der Steine gespeichert. Die Variable hilft später dabei, das Spielende zu erkennen. Die Steine werden je nach Zeile in unterschiedlichen Rottönen eingefärbt. Außerdem wird jeder Stein durch `name = "brick"` benannt. Das ermöglicht es später, alle Steine unkompliziert aus der Spielszene zu löschen.

`isDynamic = false` schließt aus, dass sich die Steine bewegen. Die weiteren `physicsBody`-Eigenschaften bewirken, dass der Ball von den Steinen abprallt und dass die `contactDelegate`-Methode `didBegin` bei jeder Kollision aufgerufen wird.

```
func setupBricks(rows: Int, cols: Int) {
  let brickw = w / CGFloat(cols+1)
  let brickh = brickw / 3
  brickcounter = rows  * cols

  for row in 1...rows {
    let brickColor =
      SKColor.init(red: 0.7 + 0.3 * CGFloat(rows - row) /
                                    CGFloat(rows),
                   green: 0.2,
                   blue: 0,
                   alpha: 1)
    for col in 1...cols {
      let brick = SKShapeNode(rectOf:
        CGSize(width: brickw * 0.95, height: brickh * 0.7))
      brick.fillColor   = brickColor
      brick.strokeColor = brickColor
      let x = brickw *  CGFloat(col)
      let y = h - brickh *  CGFloat(row+1)
      brick.position = CGPoint(x: x, y: y)
      brick.zPosition = 10
      brick.name = "brick"
      self.addChild(brick)

      brick.physicsBody =
        SKPhysicsBody(rectangleOf: brick.frame.size)
      with(brick.physicsBody!) {
        $0.isDynamic = false
        $0.categoryBitMask    = PhysCategory.brick
        $0.contactTestBitMask = PhysCategory.ball
        $0.collisionBitMask   = PhysCategory.ball
      }
    }   // Ende for col
  }     // Ende for row
}       // Ende func
```

Geschwindigkeitsüberlegungen

Die Anzahl der Spielsteine hat bei diesem Spiel einen großen Einfluss auf den Rechenaufwand. Bei jedem Frame muss SpriteKit für jeden Spielstein überprüfen, ob dieser mit dem Ball kollidiert. Das verursacht angesichts der einfachen Objektformen einen verblüffend hohen CPU-Aufwand.

Ich habe das Spiel auf einem iPhone 5S mit 16 × 16 Steinen getestet. Dabei sank die Frame-Rate auf ca. 20. Sie wurde auch nicht höher, als ich den Ball ins Aus laufen ließ, sich also sämtliche Objekte im Ruhezustand befanden. SpriteKit ist hier offensichtlich schlecht optimiert.

Wenn Sie also eine Profiversion von Breakout programmieren möchten, sollten Sie auf die Verwendung der Physik-Engine verzichten und die Kollisionsberechnungen selbst durchführen. Das ist nicht besonders schwierig und lässt sich für die simple Spielidee sicherlich effizienter gestalten, als dies in SpriteKit der Fall ist.

Schläger einrichten (Paddle)

Der Schläger ist einfach ein weiteres `SKShapeNode`-Objekt, das im untersten Bildschirmviertel platziert wird. Damit der Ball vom Schläger abprallt, ist ein physikalischer Körper erforderlich. Gleichzeitig soll der Schläger aber nicht durch die Regeln der Physik bewegt werden, sondern ausschließlich durch den Finger auf dem Display. Daher ist `isDynamic = false` erforderlich. Die `BitMask`-Einstellungen stellen sicher, dass der Ball vom Schläger abprallt, dass SpriteKit aber keine weiteren (rechenaufwendigen) Kollisionsdetektoren aktiviert.

```
func setupPaddle() {
  // Paddle unten positionieren,
  // Breite ist 1/5 der Bildschirmbreite
  paddle = SKShapeNode(
    rectOf: CGSize(width:  w * 0.20,
                   height: w * 0.04))
  paddle.fillColor   = SKColor.red
  paddle.strokeColor = SKColor.red

  paddle.position =
    CGPoint(x: w * 0.5,  // mittig
            y: h * 0.2)  // unteres Viertel
  paddle.zPosition = 10
  self.addChild(paddle)

  paddle.physicsBody =
    SKPhysicsBody(rectangleOf: paddle.frame.size)
  with(paddle.physicsBody!) {
    $0.isDynamic = false
    $0.categoryBitMask    = PhysCategory.paddle
    $0.collisionBitMask   = PhysCategory.ball
    $0.contactTestBitMask = PhysCategory.none
  }
}
```

Ball einrichten

Der Ball ist ein kreisförmiges `SKShapeNode`-Objekt. Er wird vertikal etwas oberhalb des Schlägers positioniert. Die horizontale Position ist vom Zufall abhängig.

Den physikalischen Körper wollte ich ursprünglich rund einrichten (also mit `SKPhysicsBody(circleOfRadius ...)`), das hat sich aber nicht bewährt. Es kommt dann recht oft vor, dass der Ball auf die Ecke eines Steins trifft und in einem sehr flachen Winkel abprallt. Das mag physikalisch korrekt sein, der Effekt ist aber unvorhersehbar und dem Spiel nicht dienlich. Deswegen habe ich den physikalischen Körper einfach quadratisch gewählt.

Bei der Einstellung der physikalischen Eigenschaften ist es wichtig, jede Form der Reibung oder Dämpfung zu deaktivieren – sonst wird der Ball immer langsamer. Der Ball soll von allen anderen Körpern abprallen. Eine Kollisionsbenachrichtigung ist aber nur beim Zusammentreffen mit einem Stein erforderlich.

```
func setupBall() {
  ball = SKShapeNode(circleOfRadius: w/35)
  ball.fillColor =   SKColor.black
  ball.strokeColor = SKColor.black
  ball.position =    CGPoint(x: w * 0.25 + w * 0.5 * cRnd(),
                             y: h*0.3)
  ball.zPosition = 10
  self.addChild(ball)

  ball.physicsBody = SKPhysicsBody(rectangleOf: ball.frame.size)
  with(ball.physicsBody!) {
    $0.friction = 0
    $0.angularDamping = 0
    $0.linearDamping = 0
    $0.restitution = 1
    $0.allowsRotation = false
    $0.categoryBitMask    = PhysCategory.ball
    $0.collisionBitMask   = PhysCategory.brick +
                            PhysCategory.frame +
                            PhysCategory.paddle
    $0.contactTestBitMask = PhysCategory.brick
  }
}
```

Der Ball ist anfänglich in Ruhe – er wird erst durch die Berührung des Displays in den `touches`-Methoden in Bewegung gesetzt.

37.3 Spielsteuerung

Die `touchesBegan`-Methode ist für den Start des Spiels verantwortlich. Beim ersten Spiel mit `status == .waitForStart` ist das Spielfeld bereits eingerichtet. Daher reicht es aus, den Ball mit `applyImpulse` in Bewegung zu setzen. Wenn dagegen vorher schon eine Runde gespielt wurde, müssen der Ball und die Spielsteine neu eingerichtet werden.

```
override func touchesBegan(_ touches: Set<UITouch>,
                           with event: UIEvent?)
{
  if status == .waitForStart {
    // erstes Spiel: Ball in Bewegung setzen
    ball.physicsBody!.applyImpulse(initialImpulse())
    status = .running
    label.text = ""
  } else if status == .won || status == .lost {
    // neues Spiel starten: zuerst aufräumen ...
    ball.removeFromParent()
    for ch in self.children {
      if let childname = ch.name, childname == "brick" {
        ch.removeFromParent()
      }
    }
    // ... dann das Spielfeld neu einrichten
    setupBall()
    setupBricks(rows:6, cols:6)
    ball.physicsBody!.applyImpulse(initialImpulse())
    status = .running
    label.text = ""
    self.isPaused = false
  }
}

// Startimpuls für den Ball
func initialImpulse() -> CGVector {
  return CGVector(dx: w * 0.05, dy: w * 0.05)
}
```

Schläger steuern

Um die Steuerung des Schlägers kümmert sich die `touchesMoved`-Methode. Relevant ist dabei nur die X-Koordinate des Berührungspunkts. Sie legt die Position des Schlägermittelpunktes fest. Die in `GCOperators.swift` definierte `minMax`-Funktion stellt dabei sicher, dass der Schläger nicht über den Bildschirmrand bewegt wird.

```
override func touchesMoved(_ touches: Set<UITouch>,
                           with event: UIEvent?)
{
  if touches.count != 1 { return }
  let touch = touches.first!
  let xnew = touch.location(in: self).x
  let xmin = paddle.frame.size.width/2
  let xmax = self.frame.size.width - xmin
  paddle.position =
    CGPoint(x: minMax(xnew, minimum: xmin, maximum: xmax),
            y: paddle.position.y)
}
```

`touchesEnded` und `touchesCancelled` sind gemäß den Dokumentationsvorgaben ebenfalls implementiert, enthalten aber keinen Code. Auf den Abdruck habe ich daher verzichtet.

Test, ob der Ball im Aus ist

In der `update`-Methode wird getestet, ob die Y-Koordinate des Balls kleiner ist als die des Schlägers. In diesem Fall ist der Ball im Aus, das Spiel ist verloren und wird pausiert.

Ein wenig merkwürdig ist der restliche Code: Während der Tests ist es immer wieder vorgekommen (circa einmal pro Spiel), dass der Ball von seiner üblichen diagonalen Flugbahn abwich und sich gänzlich horizontal bzw. vertikal bewegte. Ein vernünftiges Spiel ist dann nicht mehr möglich. Wenn das passiert, die X- oder Y-Komponente der Geschwindigkeit also nahe 0 ist, wird die Geschwindigkeit ganz auf 0 zurückgesetzt. Anschließend erhält der Ball wieder seinen Startimpuls. Diese Maßnahme klingt recht radikal, ist aber im Spielverlauf optisch nicht wahrnehmbar.

```
override func update(_ currentTime: TimeInterval) {
  if status != .running { return }

  // Test, ob das Spiel verloren ist
  if ball.position.y  < paddle.frame.minPoint().y {
    label.text = "Game Over"
    self.isPaused = true
    status = .lost
    return
  }

  // manchmal passiert es, dass der Ball sich nur noch
  // horizontal oder vertikal bewegt; dann muss man
  // der Physik ein wenig nachhelfen
```

```
  if abs(ball.physicsBody!.velocity.dx) <= 0.01 ||
     abs(ball.physicsBody!.velocity.dy) <= 0.01
  {
    ball.physicsBody!.velocity = CGVector.zero
    ball.physicsBody!.applyImpulse(initialImpulse())
  }
}
```

Kollisionserkennung

Wenn der Ball einen Stein trifft, wird die Methode `didBegin` des `SKPhysicsContactDelegate`-Protokolls aufgerufen. In der Methode verweist dann `contact.bodyA` oder `contact.BodyB` auf diesen Stein. Er wird einfach entfernt. (Wenn Sie möchten, können Sie hier eine nette Animation einbauen, die den Stein explodieren oder in sich zusammenfallen lässt.)

Sollte die Anzahl der Steine damit auf 0 sinken, hat der Spieler bzw. die Spielerin das Spiel gewonnen.

```
extension GameScene : SKPhysicsContactDelegate {
  func didBegin(_ contact: SKPhysicsContact) {
    var brick:SKNode!
    if contact.bodyA.categoryBitMask == PhysCategory.brick {
      brick = contact.bodyA.node
    } else if contact.bodyB.categoryBitMask == PhysCategory.brick
     {
      brick = contact.bodyB.node
    } else {
      return
    }

    // Stein entfernen
    brick.removeFromParent()

    // Spielende?
    brickcounter -= 1
    if brickcounter == 0 {
      label.text = "Congratulations!"
      self.isPaused = true
      status = .won
    }
  }
}
```

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z