Kapitel 1
Hello World!

Traditionell beginnt jedes Buch zu einer Programmiersprache mit dem Programm »Hello World!« – und auch dieser Titel ist keine Ausnahme. »Hello World« gibt eine Zeichenkette auf dem Bildschirm aus. Die eigentliche Aufgabe des Miniprogramms besteht natürlich nicht darin, Text auf dem Bildschirm anzuzeigen; vielmehr sollen Sie mit der Entwicklungsumgebung Xcode und der Syntax von Swift vertraut werden.

In diesem Kapitel zeige ich Ihnen gleich zwei Hello-World-Varianten, die im Playground bzw. als Terminal-App ausgeführt werden:

- **Playground:** Der *Playground* ist eine Testumgebung zum Ausprobieren von Swift. In den ersten Kapiteln dieses Buchs, in denen es um die Syntax von Swift geht, ist der Playground ein unverzichtbares Hilfsmittel, um mit Swift vertraut zu werden.
- **Terminal-Anwendung:** Längerfristig besteht Ihr Ziel sicherlich darin, Apps für iOS, macOS oder eine andere Apple-Plattform zu entwickeln. Leider sind selbst einfache Apps mit viel Overhead verbunden: Sie müssen sich mit den vielen Komponenten von Xcode anfreunden, sich mit der Logik eines Model-View-Controllers auseinandersetzen und diverse APIs erlernen. Für erste Experimente bietet es sich deswegen an, Programme zu entwickeln, die keine Benutzeroberfläche aufweisen, sondern in einem Terminal-Fenster unter macOS ausgeführt werden können.

Hello World für iOS und Co.

»Und wo sind die Hello-World-Versionen für iOS, macOS oder tvOS?«, werden Sie nun vielleicht fragen. Keine Sorge, auch die gibt es – aber erst in den Kapiteln des zweiten Teils des Buchs, wo es um die Grundlagen der App-Programmierung geht.

1.1 »Hello World« im Playground

Damit Sie Programme in Swift schreiben können, benötigen Sie drei Dinge:

- einen Apple-Computer, z. B. ein MacBook oder einen iMac
- eine aktuelle Version von macOS
- eine aktuelle Version von Xcode (für dieses Buch zumindest Xcode 10.2)

Genau genommen stimmen die obigen Voraussetzungen nicht mehr ganz: Apple hat Swift Ende 2015 als Open-Source-Code freigegeben. Seitdem gibt es auch eine Linux-Version von Swift. Auf diese Variante von Swift gehe ich in Kapitel 34, »Server-side Swift«, ein.

Xcode ist *die* grafische Entwicklungsumgebung (*Integrated Development Environment* = IDE) der Apple-Welt. Sie können Xcode kostenlos im App Store herunterladen und installieren. Der Platzbedarf für Xcode auf Ihrem Mac beträgt rund 7 GByte.

Apple Developer Program

Wollen Sie Ihre Apps später über Apples App Store weitergeben, ist eine Mitgliedschaft im *Apple Developer Program* erforderlich. Diese Mitgliedschaft kostet ca. 100 EUR pro Jahr und gilt für alle Plattformen gemeinsam.

Warten Sie mit der Mitgliedschaft beim Developer Program ab, bis Sie sie wirklich benötigen! Im Gegensatz zu früher ist das Developer Program keine zwingende Voraussetzung mehr, um selbst entwickelte Apps auf Ihrem eigenen iPhone oder iPad ausprobieren zu können. Die Mitgliedschaft wird erst notwendig, wenn Sie Ihre Apps im App Store weitergeben oder für macOS signieren möchten bzw. wenn Sie Spezialfunktionen wie iCloud oder In-App-Käufe ausprobieren möchten.

Den Playground starten

Der *Playground* ist ein eigener Dokumenttyp von Xcode. Im Playground können Sie Swift-Anweisungen ausführen, ohne sie in ein richtiges Programm zu verpacken. Der Playground stellt darüber hinaus weitreichende Hilfsmittel zur Codeeingabe, zur Fehlersuche sowie zur grafischen Darstellung Ihrer Daten zur Verfügung. Der Playground ist ein fantastisches Werkzeug, um Swift kennenzulernen!

Um zum Swift-Spielplatz zu gelangen, starten Sie Xcode und klicken im Startdialog auf GET STARTED WITH A PLAYGROUND. Sie müssen nun die gewünschte Plattform auswählen (iOS, macOS oder tvOS) und zwischen einem der Templates wählen (siehe Abbildung 1.1). In den folgenden Dialogen geben Sie dem Playground einen Namen und wählen das Verzeichnis, in dem die Datei gespeichert werden soll.

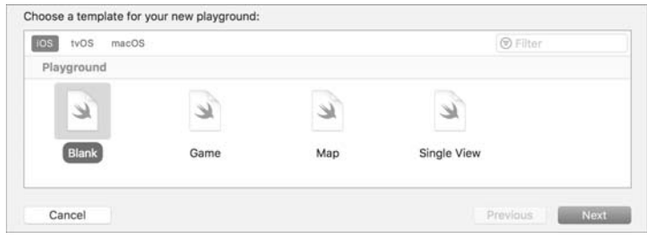

Abbildung 1.1 Einen neuen Playground einrichten

- **Plattform:** Die Plattform entscheidet, welche Bibliotheken Ihnen innerhalb des Playgrounds zur Verfügung stehen. Solange Sie nur Swift an sich ausprobieren möchten, spielt die Plattform keine Rolle.

  Die Plattformauswahl ist dann relevant, wenn Sie im Playground Steuerelemente, Bitmaps oder andere plattformspezifische Objekte erzeugen möchten. Bei einem der folgenden Beispiele ist dies der Fall – es zeigt eine Bitmap in einem macOS-spezifischen Image-View-Steuerelement an. Wenn Sie das Beispiel nachvollziehen möchten, müssen Sie sich also für einen macOS-Playground entscheiden.

- **Template:** Je nachdem, für welches Template (Muster) Sie sich entscheiden, enthält der Playground bereits Mustercode für einige Anwendungstypen. Mit gewissen Einschränkungen können Sie damit im Playground relativ fortgeschrittene Programmiertechniken ausprobieren, z. B. die Spieleprogrammierung mit der SpriteKit-Bibliothek.

  Ich bin aufgrund von Stabilitätsproblemen kein großer Fan dieser Funktion. Der Playground funktioniert fantastisch, um die Syntax von Swift zu erlernen – und dazu reicht das Template BLANK. Sobald Sie aber grafische Benutzeroberflächen, Spiele etc. gestalten möchten, sollten Sie sich vom Playground lösen und mit der Programmierung »echter« Apps beginnen.

Standardmäßig besteht das Playground-Fenster aus drei Bereichen: Im linken Bereich geben Sie den Code ein. Im grau hinterlegten rechten Bereich werden Ausgaben, Zuweisungen, die Anzahl von Schleifendurchläufen und andere Informationen angezeigt. `print`-Ausgaben erfolgen im Debug-Bereich unterhalb; dieser Bereich ist aber oft ausgeblendet.

Bemerkenswert am Playground ist, dass Sie Ihren Code nicht explizit ausführen müssen. Die Codeausführung beginnt sofort, sobald Ihre Eingaben frei von Fehlern sind. Auch nachträgliche Änderungen in weiter oben befindlichen Codezeilen werden berücksichtigt; alle Ausgaben werden sofort entsprechend aktualisiert.

Nach dem Start eines macOS-Playgrounds enthält der Codebereich bereits einen Kommentar, die Anweisung `import Cocoa` sowie die Zuweisung `var str = "Hello, playground"`. Die `import`-Anweisung ermöglicht es, Klassen der Cocoa-Bibliothek zu nutzen. Diese Bibliothek dient zur Programmierung grafischer Benutzeroberflächen. Haben Sie hingegen den Playground für iOS betreten, dann wird statt Cocoa die iOS-spezifische UIKit-Bibliothek importiert.

Hello World!

Sie können nun im Playground eigene Swift-Anweisungen eingeben. Für das klassische Hello-World-Programm ist nur eine einzige Zeile erforderlich:

```
print("Hello World!")
```

Das bedeutet, dass die Zeichenkette "Hello World!" auf dem Bildschirm ausgegeben werden soll. Tatsächlich erscheint die Ausgabe im grau hinterlegten Bereich des Playgrounds (siehe Abbildung 1.2).

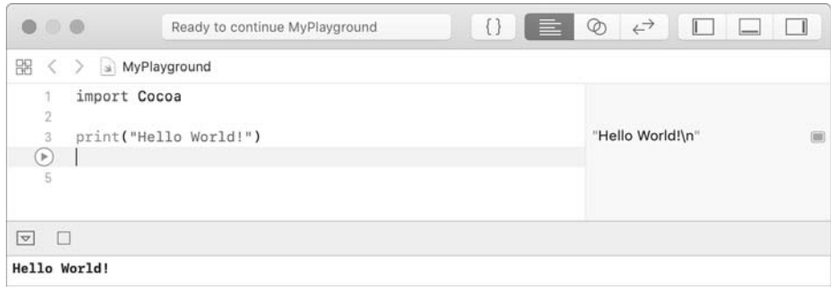

Abbildung 1.2 »Hello World!« im Playground

Im unteren Fensterbereich werden schließlich alle `print`-Ausgaben in der Reihenfolge angezeigt, in der sie durchgeführt werden. Diesen Ausgabebereich, der offiziell DEBUG AREA heißt, können Sie durch das Verschieben des grauen Trennbalkens größer oder kleiner machen oder ganz ausblenden (siehe Abbildung 1.3).

Die erste Schleife

Um sowohl Swift als auch den Playground besser kennenzulernen, geben Sie als Nächstes eine einfache Schleife ein:

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

In dieser Schleife nimmt die Variable `i` der Reihe nach die Werte 1, 2, 3 bis 10 an. `print` soll jeden dieser Werte ausgeben. Tatsächlich sind diese Ausgaben aber nur im Ausgabebereich zu sehen, während im grau hinterlegten Bereich des Playgrounds lediglich die Ausgabe 10 TIMES zu sehen ist. Das bedeutet, dass der Inhalt der Schleife zehnmal durchlaufen wurde.

In Schleifen verzichtet Xcode darauf, alle Ausgaben direkt anzuzeigen. Wenn Sie aber den Mauszeiger über die Ausgabe 10 TIMES bewegen, erscheinen zwei kleine Icons:

- Mit dem augenförmigen Icon QUICK LOOK können Sie Objekte näher ansehen, ohne dafür ein eigenes Fenster zu öffnen.
- Das rechte, runde Icon SHOW RESULT fügt hingegen unterhalb des betreffenden Codes eine Box mit den Programmausgaben ein. Anfänglich wird darin nur die letzte Ausgabe angezeigt, also der Wert 10; erst wenn Sie innerhalb der Ausgabebox den Kontextmenüeintrag VALUE HISTORY anklicken, zeigt Xcode alle Ausgaben in einer Box mit Scroll-Möglichkeit (siehe Abbildung 1.3).

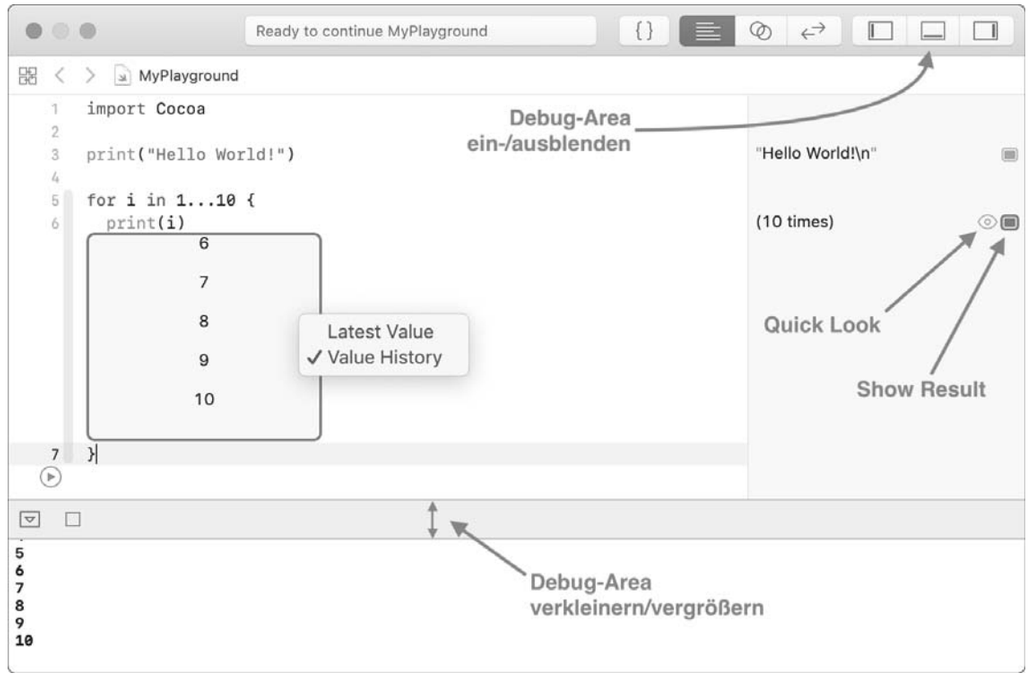

Abbildung 1.3 Playground-Darstellungsoptionen

Grafische Darstellung von Daten

Noch deutlich beeindruckender fällt die grafische Darstellung von größeren Datenmengen aus. Um das auszuprobieren, geben Sie die folgende Schleife ein:

```
for x in 1...100 {
  let y = sin(Double(x)/10)
  print("x=\(x), y=\(y)")
}
```

Hier durchläuft die Variable `x` die Zahlen 1, 2, 3 bis 100. In der Schleife wird `y` gemäß der mathematischen Formel *sin(x/10)* berechnet. Damit das funktioniert, muss die ganze Zahl `x` mit `Double` explizit in eine Fließkommazahl umgewandelt werden. Anschließend werden beide Werte ausgegeben. Das Miniprogramm nutzt die in Zeichenketten zulässige Syntax `\(ausdruck)`, um den in den Klammern befindlichen Ausdruck in die Zeichenkette einzubauen.

Klicken Sie nun bei der Ausgabe 100 TIMES neben der Zuweisung `y = ...` auf SHOW RESULT, dann werden alle Werte, die `y` im Verlauf der Schleife angenommen hat, in einer Grafik dargestellt (siehe Abbildung 1.4). Die Sinusfunktion ist darin gut wiederzuerkennen.

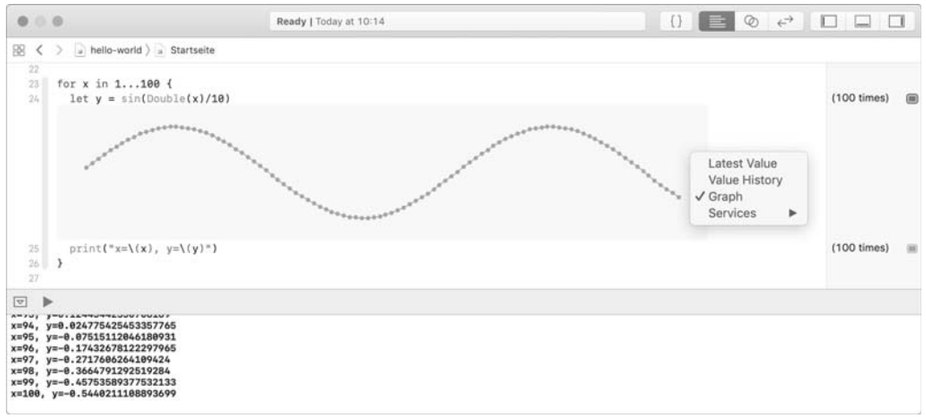

Abbildung 1.4 Grafische Darstellung von Daten

Darstellung von Objekten

Das folgende Beispiel zeigt, dass Sie im Playground auch Steuerelemente und andere grafische Objekte darstellen können.

> **macOS-Playground**
>
> Dieses Beispiel setzt voraus, dass Sie in einem Playground für macOS arbeiten! Es verwendet Klassen aus der Cocoa-Bibliothek, die in einem iOS- oder tvOS-Playground nicht zur Verfügung stehen. Es reicht auch nicht aus, die `import`-Zeile zu ändern. Sie müssen den Playground als macOS-Playground einrichten (File • New • Playground • macOS/Blank).

```
let rect = NSRect(x: 0, y: 0, width: 600, height: 400)
let v    = NSImageView(frame: rect)
let url  = URL(string: "https://kofler.info/uploads/foto.jpg")
let pic  = NSImage(contentsOf: url!)
v.image  = pic
```

Der obige Code ist schon ein wenig komplexer:

- Die Variable `rect` wird verwendet, um eine `NSRect`-Struktur zu speichern. Sie repräsentiert ein Rechteck in der Größe von 600 × 400 Pixel.
- In der nächsten Zeile wird ein `NSImageView`-Steuerelement in der Größe des `NSRect`-Objekts erzeugt. `NSImageView` ist eine Klasse der Cocoa-Bibliothek, die in macOS-Apps zur Darstellung von Bildern dient.
- Die Variable `url` verweist auf ein `URL`-Objekt mit der Adresse einer Bitmap-Datei im Web.

- In der vierten Anweisung wird diese Bilddatei aus dem Internet heruntergeladen und in einem `NSImage`-Objekt gespeichert. Das Ausrufezeichen nach dem Variablennamen `url` ist erforderlich, weil die Variable `url` ein sogenanntes *Optional* ist – also eine Variable, die auch den Zustand `nil` im Sinne von »leer«, »nicht definiert« enthalten kann. Das Ausrufezeichen bewirkt eine zwingende Umwandlung in ein `URL`-Objekt. (Sollte `url` tatsächlich `nil` aufweisen, was in diesem Beispiel nicht zu erwarten ist, tritt an dieser Stelle ein Fehler auf.)
- Im letzten Schritt wird dieses `NSImage`-Objekt schließlich als Inhalt des `NSImageView`-Steuerelements verwendet. Das ist die einfachste Möglichkeit, eine Bitmap in einem macOS-Programm darzustellen.

Wenn Sie nun im Playground neben der Ausgabe der letzten Anweisung das Icon Show Result anklicken, dann zeigt Xcode das `NSImageView`-Steuerelement samt der aus dem Internet geladenen Bitmap an (siehe Abbildung 1.5).

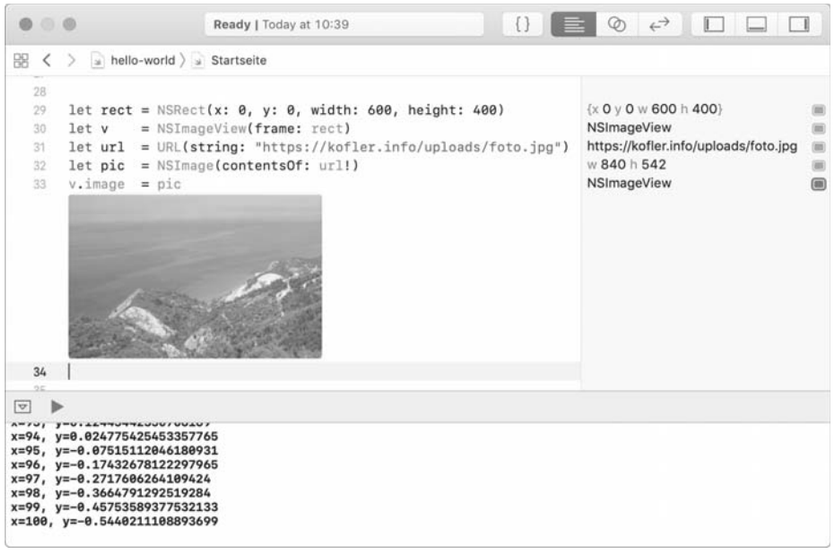

Abbildung 1.5 Darstellung von Steuerelementen im Playground

> **Was bedeutet NSxxx?**
>
> Das Kürzel NS steht für *NextStep*. NextStep war das Betriebssystem des Unternehmens NeXT, das 1996 von Apple gekauft wurde. macOS basiert auf NextStep und verwendet bis heute viele Klassenbibliotheken, die ursprünglich für NextStep entwickelt wurden. Deswegen beginnen die entsprechenden Klassennamen mit `NS`. Analog beginnen Klassennamen des UIKits, also des wichtigsten iOS-Frameworks, mit `UI`. Weitere Abkürzungen, die in diesem Buch häufig vorkommen, fasst Tabelle 1.1 zusammen.

Abkürzung	Bedeutung
CA	Core Animation
CG	Core Graphics
CK	CloudKit
CL	Core Location
GK	GameKit
IB	Interface Builder
IDE	Integrated Development Environment
MK	MapKit
MVC	Model-View-Controller
NIB	NeXT Interface Builder
NS	NextStep
OOP	objektorientierte Programmierung
REPL	Read Eval Print Loop
SF	Safari
SK	SpriteKit
UI	User Interface (UIKit-Framework)
WK	WebKit
XIB	XML Interface Builder

Tabelle 1.1 Abkürzungsverzeichnis

Kommentare

In Swift leiten Sie einzeilige Kommentare mit // ein, mehrzeilige Kommentare beginnen mit /* und enden mit */. Innerhalb des Playgrounds haben Sie darüber hinaus die Möglichkeit, Kommentare mit //: bzw. /*: einzuleiten. Damit wird nachfolgender Text gemäß der Markdown-ähnlichen *reStructuredText*-Syntax formatiert. (*Markdown* ist eine Syntax zur einfachen Auszeichnung formatierten Texts.)

```
*kursiv*, **fett**, `Listing-Schrift`

# Überschrift in Ebene 1
## Überschrift in Ebene 2
### Überschrift in Ebene 3

* Aufzählung Punkt 1
* Punkt 2
```


Das gibt Ihnen die Möglichkeit, formatierte Kommentare zu verfassen und auf diese Weise besonders gut lesbare Playground-Dokumente zu erstellen. Ob die Kommentare als Quelltext oder in formatierter Schrift angezeigt werden, stellen Sie mit EDITOR • SHOW RENDERED MARKUP bzw. SHOW RAW MARKUP ein (siehe Abbildung 1.6).

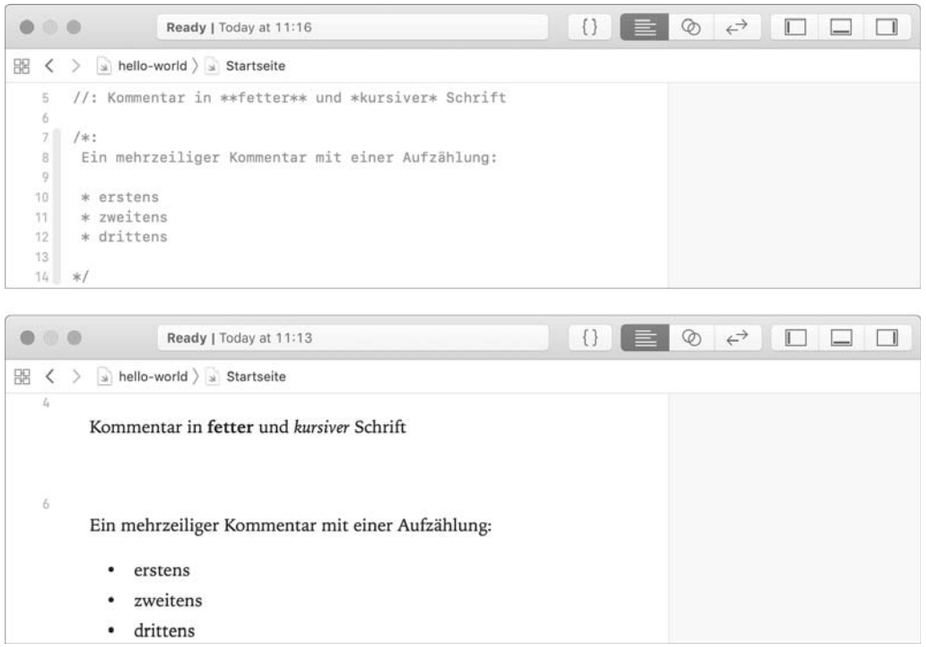

Abbildung 1.6 Oben die Quelltextansicht eines Kommentars, unten die formatierte Ansicht

> **Noch mehr Gestaltungsmöglichkeiten**
>
> Innerhalb von regulären Xcode-Projekten haben Sie in Kommentaren noch mehr Gestaltungsmöglichkeiten, die ich Ihnen in Kapitel 2, »Swift-Crashkurs«, präsentieren werde.

Playgrounds mit mehreren Dateien

Playgrounds können aus mehreren Dateien bzw. »Seiten« bestehen. Zur Verwaltung der Dateien öffnen Sie mit ⌘+1 den Navigator (siehe Abbildung 1.7) und fügen mit FILE • NEW • PLAYGROUND PAGE weitere Dateien hinzu.

Die schon vorhandene erste Seite des Playgrounds erhält dann im Navigator den Namen `Untitled Page`, die zweite Seite wird `Untitled Page 2` genannt. Natürlich können Sie diese Namen ändern. ⌘+0 blendet den Navigator wieder aus.

Abbildung 1.7 Mehrseitiger Playground, dessen aktuelle Seite eine Bitmap aus dem Verzeichnis »Resources« anzeigt

In einem weiteren Schritt können Sie die Seiten durch Querverweise verbinden. Die Links definieren Sie als Markdown-Kommentare gemäß der folgenden Syntax:

```
//: [Hier geht's zur nächsten Seite](@next)

//: [Zurück zur vorigen Seite](@previous)
```

Außerdem können Sie nun per Drag & Drop Bitmaps zum Playground hinzufügen und auch auf diese Bitmaps durch Markdown-Kommentare verweisen:

```
//: ![Bildbeschreibung](monster.png)
```

Diese Gestaltungsmöglichkeiten sind vor allem dann interessant, wenn Sie Playgrounds im Unterricht verwenden möchten.

Swift Playgrounds auf dem iPad

Apple bietet im App Store die kostenlose App *Swift Playgrounds* für iPads an. Die App soll vor allem Jugendlichen dabei helfen, die Grundkonzepte von Swift spielerisch zu erlernen. Die Animationen sprechen vermutlich eher acht- bis zehnjährige Kinder an, aber lassen Sie sich davon nicht abhalten: Die App hat wesentlich mehr Substanz, als die grafische Gestaltung vermuten lässt, und ist durchaus auch für erwachsene Einsteiger in die Swift-Programmierung interessant. In zahlreichen, allmählich anspruchsvolleren Übungen lernen Sie durchaus systematisch Methoden, Schleifen, `if`-Abfragen etc. kennen.

1.2 »Hello World« als Terminal-App

Längerfristig wollen Sie natürlich »richtige« Apps programmieren, also Programme mit grafischer Benutzeroberfläche, die unter iOS oder macOS laufen. Leider ist der Overhead recht hoch, auch wenn Sie nur eine minimale Benutzeroberfläche zusammenstellen wollen: Sie müssen sich nicht nur mit diversen Bibliotheken, Klassen und Programmiertechniken auseinandersetzen, sondern sich auch an die komplexe Xcode-Oberfläche gewöhnen.

In den folgenden Kapiteln geht es mir vorerst nur darum, Ihnen die Sprache Swift näherzubringen. Zum Ausprobieren vieler Sprachelemente ist der Playground vollkommen ausreichend. Wo dies nicht der Fall ist, bietet sich die Realisierung des betreffenden Codes als *Command Line Tool* bzw. *Terminal-App* an. Dabei handelt es sich um minimalistische Programme ohne grafische Benutzeroberfläche, die direkt in Xcode oder in einem Terminal-Fenster ausgeführt werden. Soweit diese Programme Eingaben verarbeiten oder Ausgaben durchführen, erfolgen diese im Textmodus.

Xcode kennenlernen

Auch wenn die Programmierung einer Terminal-App vergleichsweise einfach ist, müssen Sie hierfür erstmalig ein eigenständiges Xcode-Projekt erstellen. Dazu wählen Sie im Xcode-Startdialog den Eintrag CREATE A NEW XCODE PROJECT. Sollte Xcode schon laufen, starten Sie ein neues Projekt mit FILE • NEW • PROJECT.

Damit gelangen Sie nun in einen Dialog mit Vorlagen für neue Projekte (siehe Abbildung 1.8). Dort wählen Sie aus der Gruppe MACOS • APPLICATION den Eintrag COMMAND LINE TOOL aus.

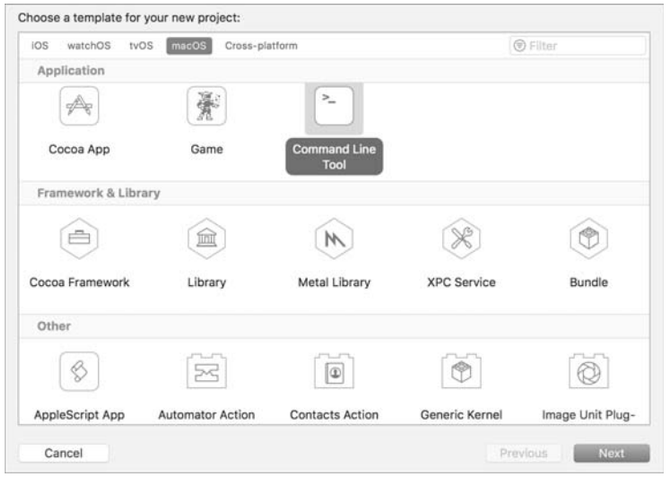

Abbildung 1.8 Dialog zum Start eines neuen Projekts

Im nächsten Schritt müssen Sie Ihrem Projekt einen Namen geben und als Programmiersprache Swift auswählen (siehe Abbildung 1.9). Product Name ist dabei der Programmname. Als Organization Identifier geben Sie üblicherweise den Domainnamen Ihrer Website in umgekehrter Reihenfolge an – also `info.kofler` für *https://kofler.info*. Wenn Sie über keine eigene Domain verfügen, verwenden Sie vorerst Ihren Nachnamen oder einfach `test`. Spätestens bei der Entwicklung von Apps, die Sie später in den App Store hochladen möchten, müssen Sie hier aber echte Daten angeben.

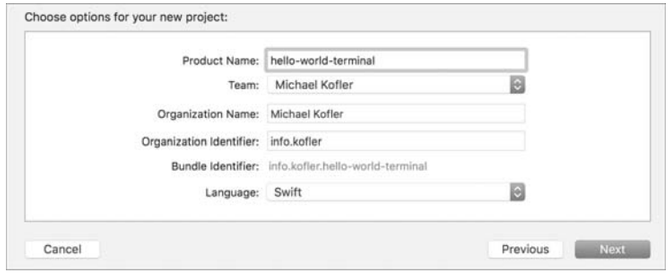

Abbildung 1.9 Eigenschaften des neuen Projekts

Im nächsten Schritt müssen Sie angeben, wo Xcode die Dateien des Projekts speichern soll. Xcode erstellt generell für jedes Projekt ein eigenes Verzeichnis, dessen Name mit dem Produktnamen übereinstimmt. Im Verzeichnisauswahldialog geht es darum, das Basisverzeichnis für die Projektverzeichnisse festzulegen. Wenn es sich bei »Hello World« um Ihr erstes Xcode-Projekt handelt, empfiehlt es sich, jetzt ein Verzeichnis für alle Ihre Xcode-Projekte anzulegen. Später können Sie in diesem Verzeichnis dann beliebig viele weitere Projekte einrichten. Bei manchen IDEs würde man dieses Basisverzeichnis als Workspace-Verzeichnis bezeichnen. Xcode verwendet diesen Begriff aber in einem anderen Zusammenhang.

Versionskontrolle mit Git

Der Verzeichnisauswahldialog enthält unten die Option Create a Git repository. Damit können Sie Ihren Code unter eine Revisionskontrolle stellen. Das gibt Ihnen die Möglichkeit, später Änderungen am Code nachzuvollziehen und bei Bedarf wieder rückgängig zu machen. Besonders attraktiv ist Git für Projekte, an denen mehrere Personen arbeiten. Für »Hello World« ist Git aber definitiv überflüssig. Eine Kurzeinführung zu den Git-Funktionen in Xcode finden Sie in Abschnitt 33.3, »Versionsverwaltung mit Git«.

Nach der Verzeichnisauswahl erscheint die eigentliche Xcode-Benutzeroberfläche (siehe Abbildung 1.10). Außer der in den Fenstertitel integrierten Symbolleiste gibt es vier Bereiche, die ich Ihnen in Abschnitt 2.6, »Xcode-Crashkurs«, im Detail vorstelle.

So viel vorweg: In der linken Spalte wählen Sie im Project Navigator die Datei Ihres Projekts aus, die Sie nun bearbeiten wollen. Dort finden Sie die Datei `main.swift` mit dem Code des Hello-World-Projekts. Klicken Sie diesen Eintrag an!

Abbildung 1.10 Der Hello-World-Code in Xcode

Die mittlere Spalte zeigt den Code bzw. den Inhalt der jeweils ausgewählten Datei an. Unterhalb des Codes kann optional der Debugging-Bereich mit den Konsolenausgaben eingeblendet werden.

Die rechte Spalte ist in mehrere Bereiche aufgeteilt, die für das Hello-World-Projekt aber allesamt nicht relevant sind. Sie können diese Spalte durch den entsprechenden Button Werkzeugleiste ausblenden in der Symbolleiste vorerst ausblenden.

Was die Codeeingabe betrifft, stellt das Hello-World-Programm eine Enttäuschung dar: Die einzig erforderliche Zeile lautet `print("Hello World!")`, und diese Zeile ist standardmäßig bereits vorhanden. Also können Sie das Miniprogramm sofort mit dem Run-Button bzw. mit ⌘+R ausführen. Das sollte auf Anhieb gelingen, allerdings kann es sein, dass Sie in Xcode die Ausgabe nicht sehen: Xcode blendet zwar den Debug-Bereich bei der Ausführung der ersten `print`-Funktion ein, dieser Bereich besteht aber wieder aus zwei Teilen. Oft ist nur der linke Teil, Variables View, sichtbar; dann müssen Sie mit dem Icon Console rechts unten im Debugging-Bereich auch den rechten Ausgabebereich einblenden.

Die Foundation

Xcode fügt in den Code neuer Terminal-Apps automatisch `import Foundation` ein. Damit können Sie in Ihrem Programm auf eine Sammlung elementarer Basisklassen zurückgreifen. Viele Klassennamen beginnen mit `NS`, weil auch die Foundation ein Erbe aus der NextStep-Vergangenheit Apples ist. Foundation-Klassen mit dem klassischen `NS`-Präfix sind unter anderem `NSArray`, `NSError` und `NSString`.

Bei vielen anderen Foundation-Klassen wurde das NS-Präfix allerdings eliminiert, z. B. bei `Date`, `Locale` und `URL` (ehemals `NSDate`, `NSLocale` und `NSURL`). Das sieht auf den ersten Blick inkonsequent aus. Warum wurde `NS` nicht gleich für alle Foundation-Klassen eliminiert?

Ein Grund besteht darin, dass man bei einigen Klassen ganz bewusst zwischen der Swift-eigenen Implementierung und der Foundation-Implementierung unterscheiden wollte, z. B. bei `String` und `NSString`. Darüber hinaus haben sich die Swift-Entwickler eine Menge Gedanken darüber gemacht, bei welchen Klassen der ursprüngliche Name erhalten bleiben sollte und bei welchen nicht. Wenn es Sie interessiert, können Sie die Logik hier nachlesen:

https://github.com/apple/swift-evolution/blob/master/proposals/0086-drop-foundation-ns.md

Bei macOS- und iOS-Programmen beginnen Codedateien mit `import Cocoa` bzw. `import UIKit`. Damit stehen im Code spezifische Klassen zur Programmierung von macOS- bzw. iOS-Oberflächen zur Verfügung. Das Foundation-Framework wird in diesen Fällen als Abhängigkeit gleich mitimportiert, d. h., Cocoa bzw. UIKit bauen auf der Foundation-Bibliothek auf.

Wo ist die App?

In der Überschrift zu Abschnitt 1.2 war ja von einer »Terminal-App« die Rede. Wo ist also diese App? Xcode speichert das neue Programm beim Kompilieren in einem Unterverzeichnis innerhalb von `/Users/ihr-login/Library/Developer/Xcode`. Am schnellsten gelangen Sie in dieses Verzeichnis, wenn Sie in Xcode im Projekt-Navigator PRODUCTS aufklappen und dann für den Eintrag HELLO-WORLD-TERMINAL das Kontextmenükommando SHOW IN FINDER ausführen.

Sie können das Programm übrigens wirklich im Terminal ausführen. Dazu öffnen Sie ein Terminal-Fenster und wechseln in das von Xcode erzeugte Debug-Verzeichnis. Am schnellsten gelingt das, wenn Sie im Terminal nur `cd` eingeben und dann das Verzeichnis per Drag & Drop vom Finder in das Terminal verschieben. Anschließend führen Sie das Programm aus, indem Sie dem Programmnamen `./` voranstellen (siehe Abbildung 1.11). Das ist notwendig, weil bei der Programmausführung im Terminal aus Sicherheitsgründen nur Programme aus bestimmten Verzeichnissen berücksichtigt werden. Deswegen müssen Sie das aktuelle Verzeichnis explizit angeben:

```
$ cd /Users/kofler/Library/Developer/Xcode/DerivedData/ \
  hello-world-terminal-dbencghqrkwmcqcslkfgwjlqllbd/Build/Products/
    Debug
$ ./hello-world-terminal
Hello, World!
```

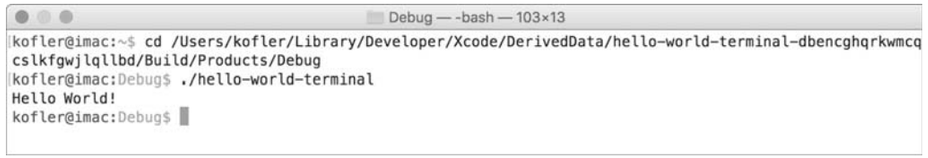

Abbildung 1.11 Programmausführung im Terminal

Momentan geht es nur darum, das Programm für Testzwecke auszuführen. Wollen Sie dagegen eine Terminal-App dauerhaft auf Ihrem Rechner installieren oder auf andere Rechner verteilen, müssen Sie ein Release-Kompilat erstellen. Die entsprechenden Kommandos und Einstellungen finden Sie im PRODUCT-Menü. Sie werden in Abschnitt 10.7, »Lottosimulator«, näher erläutert.

Mehr als nur »Hello World!«

Sollten Sie bisher mit anderen Programmiersprachen gearbeitet haben, dann wird es Sie vielleicht irritieren, dass der Programmcode in `main.swift` so minimalistisch ausfällt:

- Wo ist die `main`-Funktion oder -Methode?
- Wie können Sie auf Parameter zugreifen, die beim Aufruf an das Programm übergeben wurden?

Die erste Frage erübrigt sich: In Swift ist es nicht erforderlich, den Startcode der Terminal-App explizit in eine `main`-Funktion zu verpacken. Hinter den Kulissen kümmert sich der Compiler darum. Sie können innerhalb von `main.swift` aber natürlich andere Funktionen definieren, wie das folgende Beispiel beweist:

```
import Foundation
func sayHello(name: String)  {
    print("Hello, \(name)!")
}

// den Namen des aktiven Nutzers ermitteln
var username = NSFullUserName()

// Funktion sayHello aufrufen
sayHello(name: username)
```

Zur Beantwortung der zweiten Frage werten Sie `CommandLine.arguments` aus. Das erste Element dieses Arrays enthält immer den Programmnamen inklusive des gesamten Pfads. Wenn Parameter übergeben wurden, dann können sie aus den weiteren Array-Elementen gelesen werden. Die Anzahl der Parameter ermitteln Sie mit `args.count - 1`.

`for i in 1..<n` ist der einfachste Weg, in Swift eine Schleife zu formulieren. Dabei nimmt die Variable `i` der Reihe nach die Werte 1, 2, 3 bis zum Wert `n - 1` an. In unserem Fall soll die Schleife bis `args.count - 1` gehen.

Vielleicht wundern Sie sich, warum die Schleife nicht mit 0 beginnt. Das liegt daran, dass `args[0]` eine besondere Bedeutung hat: Das erste Element der `CommandLine.arguments` enthält immer den Programmnamen. Daran sind wir nicht interessiert; wir wollen nur die Parameter, die übergeben wurden.

```
// Zugriff auf Parameter, die an das Programm übergeben wurden
let args = CommandLine.arguments

print("Es wurden \(args.count-1) Parameter übergeben.")

for i in 1..<args.count  {
  print("Parameter \(i): \(args[i])")
}
```

Bleibt noch die Frage zu klären, wie Sie überhaupt Parameter an ein Programm übergeben. Am einfachsten gelingt das, wenn Sie Ihr Programm in einem Terminal ausführen. Alle Daten, die Sie nach dem eigentlichen Programmnamen angeben, werden als Parameter übergeben (wobei Jokerzeichen etc. vorher von der im Terminal laufenden Shell durch Dateinamen oder andere Zeichenketten ersetzt werden):

```
$ cd /Users/kofler/Library/Developer/Xcode/DerivedData/ \
  hello-world-terminal-dbencghqrkwmcqcslkfgwjlqllbd/Build/Products/
    Debug

$ ./hello-world-terminal a b c
  Hello, World!
  Hello, Michael Kofler!
  Es wurden 3 Parameter übergeben.
  Parameter 1: a
  Parameter 2: b
  Parameter 3: c
```

Für die Programmausführung innerhalb von Xcode geben Sie die gewünschten Parameter mit PRODUCT • SCHEME • EDIT SCHEME im Dialogblatt ARGUMENTS an.

Build- und Run-Schemata

Xcode verwendet Schemata, um alle Einstellungen für das Kompilieren und Ausliefern von Programmen in verschiedenen Stadien der Programmentwicklung zu verwalten. Es gibt vier vordefinierte Schemata, unter anderem für das Debugging, also für gewöhnliche Testläufe zur Fehlersuche, und für das Profiling zur Performance-Analyse. Bei Bedarf können Sie im Menü PRODUCT • SCHEME eigene Schemata definieren.

Wenn Sie den RUN-Button in der Symbolleiste von Xcode etwas länger anklicken, öffnet sich an dessen Stelle ein Menü zur Auswahl eines Schemas. Ab sofort gilt dieses Schema als Defaultschema für den Button – so lange, bis es wieder geändert wird.

Mehr Details zum Umgang mit Schemata folgen in Abschnitt 32.4, »Mehrsprachige Apps«. Dort zeige ich Ihnen, wie Sie mehrere Schemata erstellen, um eine iOS-App in unterschiedlichen Spracheinstellungen zu testen.

Den Swift-Interpreter und -Compiler direkt aufrufen

In diesem Buch steht die Entwicklung von Programmen innerhalb von Xcode im Vordergrund. Es schadet aber nicht zu wissen, dass Sie Swift auch außerhalb von Xcode auf unterschiedliche Weisen einsetzen können.

Zwar ist Swift kein Interpreter, es gibt aber einen kaum bekannten REPL-Modus (*Read Eval Print Loop*): In diesem Modus werden eingegebene Anweisungen sofort kompiliert und dann ausgeführt. Swift verhält sich also ähnlich wie ein Interpreter.

Nach dem Start von `swift` liefert `:help` einen mehrseitigen Hilfetext. Andere Eingaben, die sich auch über mehrere Zeilen erstrecken dürfen, werden sofort ausgeführt. ctrl+D beendet den Interpreter.

```
$ swift
Welcome to Apple Swift version 5.0 (swiftlang-1001.0.60.3 clang
    -1001.0.37.8).
Type :help for assistance.

  1> :help
The REPL (Read-Eval-Print-Loop) acts like an interpreter.
Valid statements, expressions, and declarations are immediately
compiled and executed ...

  1> 2+3
$R0: Int = 5

  2> print("Hello World!")
Hello World!

  3> for i in 1...3 { print(i) }
1
2
3
  4> for i in 1...3 {
  5.     print(i)
  6. }
1
2
3
<ctrl>+<D>
```

Wenn Swift also in der Lage ist, Kommandos ad hoc zu kompilieren und auszuführen, dann muss auch die Möglichkeit bestehen, ein Swift-Script ähnlich wie ein bash- oder Python-Script zu schreiben. Dazu schreiben Sie den gewünschten Code einfach in eine Textdatei, wobei die erste Zeile exakt wie folgt aussehen muss:

```
#!/usr/bin/env xcrun swift
```

Ein vollständiges Script könnte so aussehen:

```
#!/usr/bin/env xcrun swift
import Foundation
var username = NSFullUserName()
print("Hello \(username)!")
```

Bevor Sie die Datei zum ersten Mal ausführen können, müssen Sie im Terminal das Execute-Bit setzen:

```
$ chmod a+x mein-script.swift
$ ./mein-script.swift
Hello Michael Kofler!
```

Swift als bash- oder Python-Alternative?

Natürlich ist es faszinierend, dass Sie Swift-Programme ähnlich wie Shell-Scripts verfassen können, der praktische Nutzen ist aber gering: Einerseits laufen derartige Scripts nur, wenn auf dem Rechner Xcode installiert ist; diese Voraussetzung ist nur auf Entwicklerrechnern gegeben. Andererseits ist der Overhead für das Kompilieren sehr hoch und lohnt sich nur bei relativ komplexen Aufgaben. Einfache Scripts werden von bash oder Python *viel* schneller ausgeführt!

Im Terminal ist nicht nur der Swift-Interpreter zugänglich, Sie können dort auch Swift-Programme kompilieren. Dazu ermitteln Sie zuerst den Speicherort des *Software Development Kits* (SDK) für macOS. Das Kompilat erhält den gleichen Namen wie die Codedatei, aber ohne die Endung `.swift`. Es kann unmittelbar ausgeführt werden.

```
$ platf=$(xcrun --sdk macosx --show-sdk-path)
$ swiftc -sdk $platf mein-code.swift
$ ./mein-code
```

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« beim Punkt »Operators« dokumentiert:

https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html

Wie rasch Probleme auftreten können, zeigt der Vergleich eines Optionals mit `nil`: Der Compiler betrachtet `if opt!=nil` als Syntaxfehler, weil er die Anweisung im Sinne von `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:

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

Das funktioniert auch bei komplexeren Ausdrücken:

```
let (_, 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 Miniklasse `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).

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

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.

Um zu testen, ob eine ganze Zahl ohne Rest teilbar ist, können Sie anstelle der in allen Programmiersprachen üblichen `zahl % n == 0` seit Swift 5 auch `zahl.isMultiple(of: n)` verwenden:

```
let x = 15
print(x.isMultiple(of: 3))    // true
print(x.isMultiple(of: 4))    // false
```

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 Integer-Zahlen arbeiten, löst eine Division durch 0 einen Fehler aus.

Eine Besonderheit von Swift sind die Operatoren `&+`, `&-` und `&*`: Sie führen die Grundrechenarten für Integer-Zahlen 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 `--`. Später 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 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 nicht zur Verfügung
i+=1    // OK
```

Inkrement- und Dekrement-Operatoren selbst gemacht

Wenn Sie auf die Inkrement- und Dekrement-Operatoren nicht verzichten möchten, können Sie diese Operatoren unkompliziert selbst definieren:

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 Ihren 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 in Abschnitt 3.3.

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 fünf Operatoren, die Zahlenbereiche ausdrücken (siehe Tabelle 3.2). Bei den ersten beiden Varianten muss `n1` kleiner als `n2` sein, sonst ist der Ausdruck ungültig.

Operator	Interne Darstellung	Bedeutung
`n1...n2`	`CountableRange`	`n1` bis inklusive `n2` (`1...3` entspricht 1, 2, 3)
`n1..<n2`	`CountableClosedRange`	`n1` bis exklusive `n2` (`1..<3` entspricht 1, 2)
`...n2`	`PartialRangeThrough`	vom Anfang bis inklusive `n2`
`..<n2`	`PartialRangeUpTo`	vom Anfang bis exklusive `n2`
`n1...`	`CountablePartialRangeFrom`	von `n1` bis zum Ende

Tabelle 3.2 Range-Operatoren

Bei einseitigen Bereichen hängt der Anfang oder das Ende davon ab, worauf die Operatoren angewendet werden. Wenn der Bereich auf die Elemente eines Arrays angewendet wird, dann beginnt `...n2` bzw. `..<n2` mit dem ersten Array-Element (Index 0). `n1...` endet mit dem letzten Array-Element. Analog funktioniert dies bei Zeichenketten, wobei hier anstelle der Zahlen `n1` bzw. `n2` Elemente der Struktur `String.Index` verwendet werden müssen.

Normalerweise werden diese Range-Operatoren für ganze Zahlen bzw. String-Positionen verwendet. Es gibt aber auch Anwendungsfälle für Fließkommazahlen und Zeichenketten.

Intern sind die Range-Operatoren Kurzschreibweisen zur Erzeugung diverser `XxxxRange`-Strukturen:

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

Bei beidseitig geschlossenen Bereichen lauten die wichtigsten Eigenschaften `startIndex` und `endIndex`. Sie geben den Start- und Endwert des Bereichs an. `contains` überprüft, ob eine bestimmte Zahl im Zahlenbereich enthalten ist:

```
var r1 = 1..<10  // entspricht r = CountableRange<Int>(1..<10)
r1.startIndex    // 1
r1.endIndex      // 10
r1.contains(7)   // true
r1.contains(11)  // false
```

Bei einseitigen Bereichen geben `lowerBound` und `upperBound` die obere bzw. untere Grenze an:

```
let r2 = 17...    // entspricht CountablePartialRangeFrom<Int>(17)
r2.lowerBound     // 17
r2.contains(23)   // true
```

Darüber hinaus gibt es unzählige Methoden, die den Zahlenbereich verändern (`dropFirst`, `dropLast`, `reversed`) oder seine Elemente verarbeiten (`filter`, `forEach`, `map`):

```
(1...5).map { print($0 * 2) }  // Ausgabe 2, 4, 6, 8, 10
```

Range-Operatoren in Schleifen

Mit den Range-Operatoren definierte Bereiche können in Schleifen verarbeitet werden:

```
for i in 1...10 {
    print(i)  // Ausgabe 1, 2, ..., 10
}
```

Nach oben offene Bereiche sind ebenfalls als Basis für eine Schleife erlaubt – aber Vorsicht: Ohne `break` ergibt sich eine Endlosschleife!

```
for i in 10... {
  if i>20 {break}
  print(i)  // Ausgabe 10, 11, ..., 20
}
```

Nach unten offene Bereiche sind in Schleifen nicht erlaubt.

Range-Operatoren und Arrays

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

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

Anstelle des üblichen Zugriffs auf *ein* Array-Element mit `array[n]` können Sie auch einen Bereich für den Zugriff auf mehrere Elemente angeben. Die folgenden Beispiele zeigen die Anwendung aller fünf Range-Operatoren. Beachten Sie, dass das Ergebnis kein neues Array ist, sondern eine `ArraySlice`-Struktur. Beachten Sie auch, dass Array-Indizes immer mit 0 beginnen. `array[0]` ist also das erste Element, `array[...2]` meint daher alle Elemente bis zum dritten. Mehr Informationen zum Umgang mit Arrays folgen in Kapitel 10, »Arrays, Dictionaries, Sets und Tupel«.

```
let data = ["a", "b", "c", "d", "e"]  // Datentyp Array<String>
data[2...4]     // ["c", "d", "e"], Datentyp ArraySlice<String>
data[2..<4]     // ["c", "d"]
data[3...]      // ["d", "e"]
data[...2]      // ["a", "b", "c"]
data[..<2]      // ["a", "b"]
```

Range-Operatoren in switch-Konstrukten

Mit dem vorhin schon vorgestellten Operator `~=` können Sie testen, ob sich ein Wert innerhalb eines Bereiches befindet:

```
1...10 ~= 8         // true
...7 ~= 12          // false
7... ~= 12          // true
...7 ~= 7           // true
..<7 ~= 7           // false
7... ~= 8           // true
```

Analog gilt diese Schreibweise auch in `switch`-Konstruktionen (siehe Abschnitt 5.3, »Verzweigungen mit switch«):

```
let n = 21
switch n {
case ..<10:
  print("kleiner als 10")
case 10...20:
  print("zwischen 10 und 20")
case 21...:
  print("größer als 20")
default:
  print("wird nie eintreten, aber syntaktisch erforderlich")
}
```


Range-Operatoren für Fließkommazahlen

Als Vergleichsbasis für `switch` bzw. für den Operator `~=` dürfen auch Fließkommabereiche gebildet werden:

```
1.7..<2.9 ~= 2.3    // true
```

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

> **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.

Range-Operatoren für Zeichenketten

Für Vergleiche mit einzelnen Zeichen bzw. in `switch`-Konstruktionen können Sie auch Zeichenbereiche verwenden. Wie bei Fließkommazahlen ist aber eine Anwendung in Schleifen nicht zulässig.

```
"a"..."z" ~= "f"    // true
"0"..."9" ~= "x"    // false
```

Als Start- und Endpunkt der Range-Operatoren sind auch `String.Index`-Elemente erlaubt. `s[startindex...endindex]` liefert allerdings keine neue Zeichenkette, sondern einen `SubString`:

```
let s = "Hello World!"
if let startindex = s.firstIndex(of: " ") {
  let part1 = s[..<startindex]  // Datentyp String.SubSequence, das ist
  let part2 = s[startindex...]  // ein typealias für Substring

  print(part1)   // "Hello"
  print(part2)   // " World!"
}
```

Beachten Sie, dass die Start- und Endpunkte des Bereichs unbedingt als `String.Index`-Elemente formuliert werden müssen. Der in nahezu allen anderen Programmiersprachen übliche Zugriff über ganze Zahlen ist in Swift nicht erlaubt (sieht auch Kapitel 8, »Zeichenketten«).

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 wei-

teren `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 3.3). 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 3.3 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 auf der folgenden Seite nachlesen:

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

3.5 Eigene Operatoren

Swift bietet Ihnen 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 3.3). 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 comparison-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
```

Neuer Unwrap-or-die-Operator

Erica Sadun hat in der Swift-Evolution-Mailing-Liste vorgeschlagen, die Sprache Swift um den Operator `!!` zu erweitern, der das Auspacken eines Optionals erzwingt. Gelingt dies nicht, wird ein Fehler ausgelöst. Die Anwendung dieses Operators sieht so aus:

```
let result = [1, 2, 3]
let firstresult = result.first !! "Fehlermeldung"
print(firstresult)
```

Es gibt also einen optionalen Ausdruck (hier `first`, liefert `nil`, wenn das Array leer ist), und Ihr Code verlässt sich darauf, dass der Ausdruck ein Ergebnis liefert. Sie könnten nun einfach durch `first!` das Auspacken erzwingen; sollte aber doch ein Fehler auftreten, dann haben Sie keine Fehlermeldung. Mit der Konstruktion `!! "Fehlermeldung"` können Sie hingegen auf die Ursache des Problems hinweisen. (Der Operator `!!` ist Ausdrucks des Frusts, den viele Swift-Entwickler im Umgang mit Optionals und der erforderlichen Absicherung durch `guard`, `if let` etc. empfinden. Mehr Informationen zu diesem Thema folgen in Abschnitt 4.2, »Optionals«, sowie in Kapitel 13, »Fehlerabsicherung«.)

Die Swift-Entwicklergemeinde hat sich nicht dazu durchringen können, den Operator zum Swift-Sprachumfang hinzuzufügen. Aber das können Sie mit wenigen Zeilen selbst erledigen:

```
// Projekt unwrap-or-die-operator, Datei main.swift
infix operator !!: NilCoalescingPrecedence
extension Optional {
  // Idee: Erica Sadun 2017
  public static func !!(optional: Optional,
                        errorMessage: @autoclosure () -> String)
    -> Wrapped
  {
    if let value = optional { return value }
    fatalError(errorMessage())
  }
}
```

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 42

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 42.1 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. Trotzdem 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 28, »SpriteKit«): Die hier präsentierte Version umfasst nur rund 250 Zeilen, wenn man den Begleitcode wegrechnet (`CGOperators.swift`, `GameViewController.swift` etc.).

42.1 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 Vollbildmodus auf iPhones im Portrait-Modus gespielt werden. Andere Geräteausrichtungen wurden in den Projekteinstellungen deaktiviert.

In Abbildung 42.1 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

42.2 Initialisierung

Im View-Controller wird die Spielszene geladen. Die Datei `GameScene.sks` enthält nur den orangefarbenen Hintergrund des Spiels. In `viewDidLoad` wird die Größe der Szene eingestellt. Dabei wird die Größe des jeweiligen iOS-Geräts aus dem `SKView`-Objekt übernommen.

Die eigentliche Initialisierung des Spiels erfolgt durch den Aufruf von `setup` in der Methode `viewDidAppear`. Vielleicht fragen Sie sich, warum die Initialisierung nicht wie sonst üblich bereits in `viewDidLoad` erfolgt. Das hat mit den Eigenheiten der aktuellen iPhone-Modelle zu tun: Beim Einrichten des Spiels muss der oberste Displaybereich mit der Notch unbenutzt bleiben. Wie viel Platz zur Verfügung steht, kann aber erst mit dem Aufruf von `viewDidAppear` durch die Auswertung der View-Eigenschaften `layoutMargins` oder `safeAreaLayoutGuide` ermittelt werden.

```
// Projekt ios-breakout, Datei GameViewController.swift
class GameViewController: UIViewController {
  // Szene aus GameScene.sks laden (oranger Hintergrund)
  var scene: GameScene! = GameScene(fileNamed: "GameScene")!
  // die Initialisierung nur einmal durchführen
  var setupDone = false

  override func viewDidLoad() {
    super.viewDidLoad()
    // Größe der Szene einstellen, Szene anzeigen
    let skView = self.view as! SKView
    scene.scaleMode = .aspectFill
    scene.size = skView.frame.size
    skView.presentScene(scene)
  }

  // wenn die Game-View zum ersten Mal erscheint, Initialisierung
  // durchführen
  override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    if setupDone { return }
    scene.setup()
    setupDone = true
  }
}
```

Control Center verbergen

Bei älteren iOS-Geräten können Sie durch 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 {
  var paddle: SKShapeNode!   // Objekte
  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 setup() eingestellt
  var brickcounter = 0   // Spielstatus
  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

Die Initialisierung der Spielszene erfolgt in der Methode `setup`, die in der `GameViewController`-Klasse durch `viewDidAppear` aufgerufen wird. Die sonst übliche Initialisierung in `didMove(to:)` ist bei dieser App nicht möglich, weil zu diesem Zeitpunkt die Eigenschaften `layoutMargins` bzw. `safeAreaLayoutGuide` noch nicht ausgewertet werden können. Diese Eigenschaften werden aber gleich mehrfach benötigt, um die Spielszene so zu gestalten, dass sie auch auf aktuellen iPhone-Modellen korrekt dargestellt wird:

- Die Höhe des Spielbereichs (Variable `h`) wird im Vergleich zur Displaygröße um `layoutMargins.top` reduziert, also um den Bereich des Displays, der abgerundet ist. (Bei älteren iPhone-Modellen enthält `layoutMargins.top` den Wert `0`. Das Spiel nutzt dann das gesamte Display.)
- Der Bereich, in dem sich der Ball frei bewegen kann, wird durch ein `SKPhysicsBody`-Objekt eingegrenzt, das der Größe der sogenannten *Safe Area* entspricht (siehe Abbildung 42.2).
- Damit für die Spieler klar ist, wo der Ball oben reflektiert wird, wird der obere Displaybereich durch ein schwarzes `SKShapeNode`-Objekt dargestellt. In Abbildung 42.2 ist dieses Objekt `wall` in grauer Farbe dargestellt. Bei anderen iPhone-Modellen entfällt dieses Objekt, weil `layoutMargins.top` dann den Wert `0` zurückgibt.


```
// Projekt ios-breakout, Datei GameScene.swift
func setup() {
  w = self.frame.width
  h = self.frame.height - self.view!.layoutMargins.top

  // 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()

  // Ball einsperren
  let gameArea = self.view!.safeAreaLayoutGuide.layoutFrame
  self.physicsBody =  SKPhysicsBody(edgeLoopFrom: gameArea)
  with(self.physicsBody!) {
    $0.categoryBitMask    = PhysCategory.frame
    $0.collisionBitMask   = PhysCategory.ball
    $0.contactTestBitMask = PhysCategory.none
  }
```

```
  // optische Abgrenzung nach oben (für aktuelle iPhone-Modelle!)
  let wallrect = CGRect(x: 0,
                        y: h,
                        width: w,
                        height: self.view!.layoutMargins.top)
  let wall = SKShapeNode(rect: wallrect)
  wall.fillColor = .black
  wall.strokeColor = .black
  wall.zPosition = 2
  self.addChild(wall)
}
```

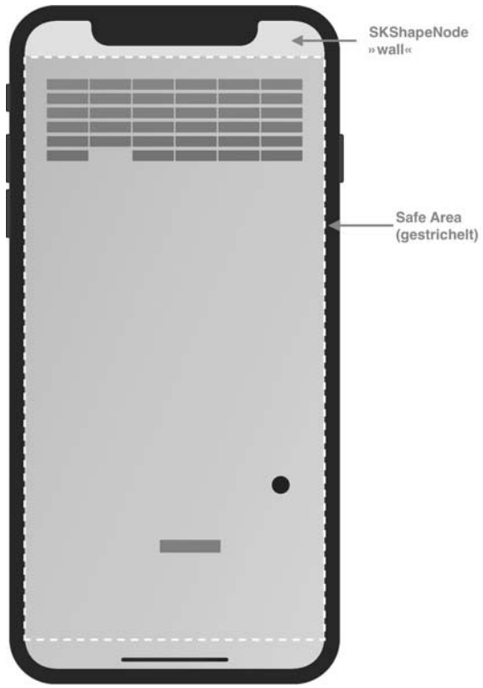

Abbildung 42.2 Die »Safe Area« auf einem aktuellen iPhone-Modell

Um bei der Einstellung der Eigenschaften des physikalischen Körpers die wiederholte Nennung von `brick.physicsBody!` zu vermeiden, habe ich die `with`-Funktion verwendet, die ich schon in Abschnitt 28.7, »Kollisionserkennung«, vorgestellt habe. 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 trotz 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, also alle Objekte im Ruhezustand waren. 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 in der Mitte
  ball.position = CGPoint(x: w * CGFloat.random(in: 0.25...0.75),
                          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.

42.3 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
    }
  }
}
```