Kapitel 1

Einführung in Swift

In diesem Kapitel

Swift ist eine neue Programmiersprache, die von Apple entwickelt und Entwicklern auf der WWDC 2014 vorgestellt wurde. Mit Swift kann man Applikationen für die Apple-Plattformen iOS und OS X entwickeln. Von seinem Design her arbeitet Swift mit Objective-C zusammen und soll es später mal ersetzen. Mit der Sprache Objective-C wurden ursprünglich die Applikationen für Apple-Plattformen erstellt.

Für Swift legt Apple verschiedene Ziele fest:

- Die App-Entwicklung soll vereinfacht und moderne Spracheigenschaften sollen integriert werden.
- Indem die häufigsten Programmierfehler unterbunden werden, ist der geschriebene Code sicherer.
- Der Code soll einfach zu lesen sein und eine klare und ausdrucksstarke Syntax enthalten.
- Swift soll kompatibel sein zu vorhandenen Objective-C-Frameworks wie Cocoa und Cocoa Touch.
- In diesem Kapitel stellen wir die grundlegende Syntax von Swift vor und legen damit das Fundament für das restliche Buch.
- Dies sind die zentralen Kapitelthemen:
- Sie verwenden `var`, um eine Variable zu deklarieren, und `let` für eine Konstante.
- Sie führen Code bedingungsabhängig mit `if`- oder `switch`-Konstrukten aus.
- Sie wiederholen Codesegmente, indem Sie mit `for`-, `for-in`-, `while`- und `do-while`-Schleifenkonstrukten arbeiten.
- Die grundlegenden Datentypen werden als `structs` implementiert, die als Wert im Code übergeben werden.
- Weil es sich bei den Basistypen um `structs` handelt, können weitere Eigenschaften oder Methoden zur Verfügung stehen.
- Arrays und Dictionaries sind bei Swift leistungsfähigere Collection-Typen als ihre Entsprechungen bei Objective-C.

1.1 Grundlagen der Syntax

Wenn Sie eine neue Sprache lernen, steht am Anfang höchstwahrscheinlich das altbekannte »Hello World«-Beispiel. In Swift besteht dieses Programm aus nur einer Zeile:

```
println("Hello World")
```

Als Erstes sollte Ihnen auffallen, was Sie nicht sehen. Der Code springt direkt zum Kern des Programms. Sie brauchen vor Beginn gar nichts weiter einzurichten. Sie müssen keine Standardbibliothek einbinden oder importieren oder eine initiale `main()`-Funktion schreiben, die vom System aufgerufen werden soll, und Sie brauchen noch nicht mal ans Ende jeder Zeile ein Semikolon zu schreiben.

Hinweis

Kommentare werden bei Swift genauso geschrieben wie bei Objective-C – mit einer wesentlichen Ergänzung: Sie können // für einen einzeiligen Kommentar nehmen oder mit /* */ einen mehrzeiligen Kommentar einfassen. Anders als bei C-basierten Sprachen dürfen Sie bei Swift Kommentare verschachteln. Das ist sehr praktisch, wenn Sie einen ganzen Codeabschnitt auskommentieren wollen, in dem bereits mehrzeilige Kommentare stehen. In diesem Buch arbeiten wir in den Beispielen mit //-Kommentaren, um Ergebnisse aufzuführen oder Erläuterungen zu ergänzen.

1.1.1 Variablen und Konstanten

Ein Programm, das nur eine statische Textzeile ausgibt, ist nicht sonderlich hilfreich. Damit ein Programm nützlich ist, muss es mit Daten arbeiten können und dafür Variablen und Konstanten verwenden. Wie der Name schon impliziert, enthalten Variablen Inhalte, die sich während der Codeausführung ändern können, während Konstanten nach ihrer Initialisierung nicht verändert werden können. Von Variablen spricht man als veränderlich (*mutable*), und Konstanten sind unveränderlich (*immutable*).

Bei Swift deklarieren Sie eine Variable mit dem Schlüsselwort `var` und eine Konstante mit `let`. Das gilt in Swift für alle Datentypen – anders als in Objective-C, wo der Typ selbst erkennen lässt, ob er veränderbar ist oder nicht, so wie bei `NSArray` und `NSMutableArray`. Bei Swift hat die veränderbare Version eines Objekts den gleichen Typ wie die unveränderliche Version und ist keine Unterklasse.

Für das restliche Kapitel gilt das über Variablen Gesagte gleichermaßen für Konstanten (vorausgesetzt, es geht nicht um die Veränderung von Daten).

Hinweis

Bevor wir uns näher mit bestimmten Typen beschäftigen, müssen wir vor allem verstehen, dass bei Swift jeder Datentyp als eine von drei unterschiedlichen Datenstrukturen implementiert wird. Jeder Typ ist entweder `enum`, `struct` oder `class`, und dafür sind dann jeweils bestimmte Eigenschaften und/oder Methoden verfügbar. Darauf werden wir im weiteren Verlauf näher eingehen, aber Sie sollten sich unbedingt von Anfang an klarmachen, dass es Unterschiede gibt, wie diese Datenstrukturen im Code jeweils übergeben werden.

Swift befolgt einige Standardregeln für den Umgang mit `enum`, `struct` und `class`. Sobald irgendwo `enums` oder `structs` übergeben werden, werden sie *als Wert übergeben* (engl. *passed by value*), d.h., es wird eine Kopie des Originals erstellt und diese dann der neuen Variablen zugewiesen. Somit kann man die neue Variable verwenden, modifizieren oder löschen, ohne dass es sich aufs Original auswirkt. Das Umgekehrte gilt ebenso: Das Original kann verwendet, verändert oder gelöscht werden, ohne dass es sich auf die neue Kopie auswirkt.

Wenn hingegen irgendwo eine `class` übergeben wird, wird sie *als Referenz übergeben* (engl. *passed by reference*), d.h., der neuen Variablen wird ein Zeiger (*pointer*) auf die ursprüngliche Variable übergeben. Wird dann bei einer der Variablen etwas geändert, wirkt sich das jeweils auch auf die andere aus.

Alle hier vorgestellten Basistypen werden hinter den Kulissen als `structs` implementiert und somit immer kopiert und dann als Wert übergeben. Weil es sich um `structs` handelt, ist es außerdem möglich, zusätzliche Funktionalitäten über Eigenschaften und/oder Methoden zu implementieren, die von den Objective-C-Gegenstücken nicht bekannt sind.

Swift ist eine stark typisierte Sprache, was bedeutet, dass jede Variable beim Kompilieren auf einen bestimmten Typ gesetzt wird und während ihrer Lebenszeit nur Werte dieses Typs enthalten kann.

Zwei häufige Typen sind `Int` und `Float` (Näheres folgt gleich). Wenn Sie eine Variable des Typs `Int` setzen, kann sie einzig `Int`-Werte speichern. Man kann sie nicht zum Speichern eines `Floats` zwingen. Typen können nie implizit in andere Typen konvertiert werden. Das bedeutet beispielsweise, dass man ein `Int` nie zu einem `Float` addieren kann. Wenn Sie zwei Zahlen addieren wollen, müssen Sie erst darauf achten, dass sie vom gleichen Typ sind, oder explizit die eine in den anderen Typ konvertieren. Unter anderem deswegen ist Swift auch so eine sichere Sprache: Der Compiler verhindert, dass Sie Typen vermischen und möglicherweise unerwartete Resultate produzieren.

Um zu erkennen, welche Gefahren beim Mischen von Typen auftreten, schauen wir uns diesen C-Code an:

```
int intValue = 0;
float floatValue = 2.5;
int totalValue = intValue + floatValue;
```

Dieser Code addiert ein `int` und ein `float`. Was würde `total` hier ergeben? Weil die Summe ein `int` ist, kann der Dezimalteil der `floatValue`-Variablen nicht gespeichert werden. `floatValue` muss zuerst implizit in ein `int` konvertiert werden, bevor es zu `intValue` addiert und in `totalValue` gespeichert werden kann. Erwartet in diesem Fall der Entwickler vom Compiler, dass dieser `floatValue` auf `3` aufrundet, oder erwartet er, dass der Dezimalteil einfach wegfällt und stattdessen `2` addiert wird? Swift vermeidet diese Mehrdeutigkeit, indem hier ein Compile-Time-Fehler ausgegeben wird. Das zwingt Sie, genau zu sagen, was geschehen soll. Auf solche Weise vermeidet Swift übliche Programmierfehler.

Sie müssen Variablen und Konstanten Namen geben, um im Code darauf verweisen zu können. Namen in Swift können aus fast allen Zeichen bestehen, z.B. auch Unicode-Zeichen. Zwar ist es möglich, dafür auch Emojis und ähnliche Zeichen zu nehmen, aber das sollten Sie eigentlich lieber lassen. Hier ist der minimal erforderliche Code zum Deklarieren einer Variablen:

```
var itemCount: Int
```

Mit diesem Code wird eine Variable namens `itemCount` vom Typ `Int` deklariert. Eine Variable muss auf einen Anfangswert gesetzt werden, bevor sie verwendet werden kann. Das können Sie machen, sobald die Variable deklariert wird:

```
var itemCount: Int = 0
```

Das geht auch später noch, allerdings spätestens dann, wenn Sie versuchen, den Wert zu lesen.

Bei Swift gibt es ein Feature namens *Type Inference* bzw. Typinferenz. Wenn der Compiler aus dem von Ihnen gesetzten Anfangswert genug Informationen bekommt, braucht der Variablentyp nicht extra deklariert zu werden. Wenn es sich bei Ihrer Variablen beispielsweise um ein `Int` handelt, deklarieren Sie das wie folgt:

```
var itemCount = 0
```

Da `0` ein `Int` ist, muss `itemCount` ebenfalls ein `Int` sein. Das ist genau das Gleiche wie beim vorigen Beispiel. Der Compiler generiert exakt den gleichen Maschinencode.

Wenn der Anfangswert der Variablen auf den Rückgabewert einer Funktion gesetzt wird, folgert der Compiler, dass es sich um den gleichen Typ wie beim Rückgabewert handelt.

Nehmen wir die Funktion `numberOfItems()`, die ein `Int` zurückgibt und die folgende Zeile:

```
var itemCount = numberOfItems()
```

dann folgert der Compiler, dass `itemCount` vom Typ `Int` ist.

Weil der Compiler exakt den gleichen Code generiert, egal ob Sie den Typ explizit angeben oder mit Typinferenz arbeiten (damit der Compiler das für Sie erledigt), hat zur Laufzeit keine der beiden Methoden einen Vorteil. Wenn Sie hingegen den Typ explizit angeben müssen, bleibt Ihnen keine Wahl. Doch falls der Compiler den Typ erkennen kann, können

Sie entscheiden, ob der Compiler das für Sie erledigt oder ob Sie sich selbst darum kümmern. Bei dieser Entscheidung gibt es zwei Dinge zu beachten. Erstens die Lesbarkeit: Wenn Sie mit Typinferenz arbeiten und der Variablentyp auch für spätere Leser des Codes völlig klar ist, dann sollten Sie sich auf jeden Fall Tipparbeit ersparen und Typinferenz einsetzen. Wenn der gesetzte Anfangswert der Rückgabewert einer seltener vorkommenden Funktion ist, ist es für spätere Leser wohl einfacher nachvollziehbar, wenn Sie den Typ explizit angeben. Sie sollten später beim Lesen des Codes nicht danach suchen müssen, was eine Funktion zurückgibt, nur um den Typ der Variablen bestimmen zu können.

Der zweite Grund für die explizite Angabe des Typs ist die zusätzliche Sicherheitsprüfung. So wird gewährleistet, dass der für eine Variable erwartete Typ und der angegebene Typ auch zueinander passen. Ist das nicht der Fall, wird ein Compile-Time-Fehler gemeldet, und Sie korrigieren das dann entsprechend.

1.1.2 String-Interpolation

Sie wissen bereits, wie man eine Textzeile über den Befehl `println` auf der Konsole ausgibt. Mit der String-Interpolation ergänzen Sie die Ausgabe mit Variablen, Konstanten und anderen Ausdrücken. Das machen Sie, indem Sie Variablen und Ausdrücke in Klammern setzen und durch einen Backslash maskiert direkt in das String-Literal einfügen:

```
var fileCount = 99
println("Es sind \(fileCount) Dateien in Ihrem Ordner")
//Ausgabe: Es sind 99 Dateien in Ihrem Ordner
```

Dies gilt nicht nur für `println`, sondern funktioniert überall, wo ein String-Literal verwendet wird:

```
var firstName = "Geoff"
var lastName = "Cawthorne"
var username = "\(firstName)\(lastName)\(arc4random() % 500)"
//Username: GeoffCawthorne253
```

1.1.3 Kontrollfluss

Sogar die einfachsten Programme benötigen irgendeine Logik, um zu bestimmen, welche Aktionen unternommen werden sollen. Das Programm muss anhand seiner vorliegenden Informationen Entscheidungen treffen. Solche Logik wie »Kommt dies vor, mache das« oder »Führe etwas *x* Mal aus« bestimmen den Fluss einer App und somit ihr Ergebnis.

Bedingungsanweisungen

Swift bietet sowohl `if`- als auch `switch`-Konstrukte, damit Sie Code bedingungsabhängig ausführen können.

Kapitel 5

Generics

In diesem Kapitel

Zur generischen Programmierung gehört die typunabhängige Definition von Algorithmen, wobei die tatsächlich involvierten Typen später bei Verwendung des Algorithmus spezifiziert werden. Durch die Programmierung mit Generics, also generischen Typen, schaffen Sie generische Templates (Vorlagen) für Funktionen und/oder Daten, indem die Funktionalität von spezifischen Typen abstrahiert wird. So können Sie sich darauf konzentrieren, weniger Code zu schreiben, der zudem leistungsfähig und einfach zu pflegen ist. Generics sind neu für Objective-C-Entwickler und ein wichtiges Konzept bei Swift. Swift setzt generische Funktionen und Typen ein, um duplizierten Code zu reduzieren und trotzdem die starken Typanforderungen der Sprache zu verwenden, um sicher zu sein, dass die Variablen immer vom erwarteten Typ sind. Mit Generics konzentrieren Sie sich darauf, wie Sie mit Ihren Daten interagieren, indem Sie definieren, welche Art von Daten Sie brauchen, und den Compiler dann den benötigten typspezifischen Code erstellen lassen. Sie treffen Annahmen über Daten und agieren basierend auf diesen Annahmen, ohne noch mühevoll die Korrektheit zu prüfen, da der Compiler das garantiert. Haben Sie z.B. ein Array, das `Int`s enthalten soll, können Sie auch davon ausgehen, weil das Array so definiert wurde, dass es nur `Int`s akzeptiert. Das Gleiche gilt für die von Ihnen definierten generischen Typen.

Dies sind die zentralen Themen dieses Kapitels:

- Mit generischen Funktionen erstellen Sie Prototypfunktionen, um den für verschiedene Typen duplizierten Code zu reduzieren.
- Durch Typparameter weiß der Compiler, welche Typen für Ihre Generics gültig sind.
- Sie erstellen generische Typen für komplexe Datenstrukturen, die auf einfacheren Typen aufbauen.
- Sie erweitern generische Typen auf gleiche Weise wie andere Typen.
- Durch verknüpfte Typen definieren Sie Protokolle, die mit einer größeren Bandbreite von Typen arbeiten können und trotzdem die Typsicherheit bewahren.
- Sie können eine `where`-Klausel bei den Typparametern einfügen, um kompliziertere Anforderungen zu erfüllen, wie z.B. den Typ aufgrund seiner verknüpften Typen zu beschränken.

5.1 Warum Generics?

Swift ist eine stark typisierte Sprache, und das bedeutet, Sie müssen den Typ aller verwendeten Variablen und Parameter angeben. Eine Funktion, die zwei `Int`-Werte akzeptiert und deren `sum` zurückgibt, kann nicht zwei `UInt`s akzeptieren und addieren, weil `Int` und `UInt` verschiedene Typen sind. Eine mögliche Lösung für dieses Problem ist, die Funktion durch Angabe zweier separater Funktionen gleichen Namens zu überladen, von denen die eine `Int`s verwendet und die andere `UInt`s. Das ist natürlich lästig und wird schnell unpraktisch, wenn die Zahl der Typen steigt, mit denen die Funktion arbeiten kann. Schon in einem einfachen Fall müssten Sie sich um die 8-, 16-, 32- und 64-Bit-Versionen von `Int` kümmern – jeweils mit und ohne Vorzeichen – insgesamt also um mindestens acht Funktionen. Swift löst die Erstellung generischer Funktionen eleganter, indem der Compiler mit prototypischen Funktionsdefinitionen die Vielzahl überladener Funktionen automatisch bei Bedarf erstellen kann.

5.2 Generische Funktionen

Die Syntax einer generischen Funktion ähnelt der einer normalen Funktionsdeklaration, enthält aber zusätzlich eine Liste von Typparametern, mit denen die Funktion arbeiten kann.

Folgendes Beispiel ist eine generische Funktion, die einen beliebigen Typ als einzigen Parameter akzeptiert:

```
func isASubclassOfNSObject<T>(objectToTest: T) -> Bool {
    return objectToTest is NSObject
}
```

Nach dem Funktionsnamen folgt in spitzen Klammern die Parameterliste : `< >`. Dieser Liste entnimmt der Compiler, welche der Typen in der restlichen Funktion Platzhalter für echte Typen sind, die später beim Aufruf der Funktion geliefert werden. In diesem Fall verwenden Sie einen Typparameter namens `T`. Der Rest der Funktion besagt, dass Sie einen Parameter namens `objectToTest` vom Typ `T` haben (der eigentliche Typ wird später bestimmt), und dann wird ein boolescher Wert zurückgegeben.

Die Implementierung der Funktion testet, ob der gelieferte Parameter eine Unterklasse von `NSObject` ist, und gibt im Wahrheitsfall `true` zurück und anderenfalls `false`:

```
println(isASubclassOfNSObject(String()))
//Ausgabe: true
println(isASubclassOfNSObject(NSString()))
//Ausgabe: true
println(isASubclassOfNSObject([Int]))
//Ausgabe: false
```

```
println(isASubclassOfNSObject(NSArray()))
//Ausgabe: true
println(isASubclassOfNSObject([String:Int]))
//Ausgabe: false
println(isASubclassOfNSObject(NSDictionary()))
//Ausgabe: true
```

Beachten Sie, dass Sie nur eine einzige Funktion deklarieren, diese aber trotzdem mit mehreren Werttypen aufrufen können. Sie erkennen bereits, wie eine einfache generische Funktion die Menge Code reduziert, die Sie schreiben und pflegen müssen.

5.2.1 Typparameter

Das vorige Beispiel akzeptiert jedes Objekt als Parameter, aber in der Praxis sollten Sie einschränken, welche Typen für Ihre Funktion akzeptabel sind. Das erreichen Sie, indem Sie die Typparameter mit Constraints versehen. So wie in der Syntax, mit der Sie anzeigen, dass ein Objekt sich an ein Protokoll halten wird, hängen Sie an den Platzhalter für den Typ einen Doppelpunkt an und schreiben danach das Protokoll, nach dem sich der Typ richten muss:

```
func sumValues<T: IntegerType>(value1:T, value2:T) -> T {
    return value1 + value2
}
```

Hier ergänzen Sie `T` mit dem Constraint, das Protokoll `IntegerType` zu befolgen. Jede der 8-, 16-, 32- oder 64-Bit-Versionen von `Int`, ob mit oder ohne Vorzeichen, wird akzeptiert und kann von der Funktion verarbeitet werden. `String`s und andere Nicht-`Int`-Typen, die an diese Funktion übergeben werden, generieren nun Compile-Time-Fehler, weil sie die spezifischen Anforderungen nicht erfüllen.

5.2.2 Mehr als einen Typparameter verwenden

Im vorigen Beispiel haben Sie nur einen Typparameter verwendet, den Sie `T` genannt haben. Sie können auch mehr als einen angeben, indem Sie die Parameter durch Kommas getrennt auflisten. Sie können den Typen auch aussagekräftigere Namen geben, um die Lesbarkeit des Codes zu verbessern:

```
func existingOrDefaultValue<KeyType:Hashable,
➥ValueType>(dict:[KeyType:ValueType], key:KeyType,
➥defaultValue:ValueType) -> ValueType {
    if let existingValue = dict[key] {
        return existingValue
    }
    return defaultValue
}
```

Mit dieser generischen Funktion erhalten die Typparameter sprechendere Namen, damit zukünftige Leser des Codes schneller verstehen, wofür die Typen verwendet werden.

5.3 Generische Typen

Mit Swift können Sie generische Typen erstellen. Ein generischer Typ ist eine neue Datenart, die auf einem oder mehreren anderen Typen aufsetzt, aber selbst ein völlig anderer Typ ist. Sie kennen bereits die beiden wichtigen generischen Typen `Array` und `Dictionary`. Wie bereits gesehen sind `Array` und auch `Dictionary` darauf angewiesen, dass ihre Typen definiert werden, da es sich um generische Typen handelt. Ein `Array` aus `Int`s ist ein völlig anderer Typ als ein einzelner `Int` und auch ein ganz anderer Typ als ein `Array` aus `String`s. Dictionaries sind in Swift generische Typen, die auf zwei Typen aufsetzen: einem für den Schlüssel (der das Protokoll `Hashable` befolgen muss) und einem für den gespeicherten Wert (muss nicht protokollkonform sein).

Schauen wir uns an, wie man einen generischen Typ erstellt, um ein Datenkarussell (»carousel«) zu erzeugen. Diese Datenstruktur nimmt Elemente in einem endlosen Kreislauf auf. Sobald ein Element hinzugefügt wurde, steht immer auch ein Nachfolger zur Verfügung, da jedes Element beim Erreichen des Endes wieder an den Anfang rutscht. Ein solches Datenkarussell ist praktisch, um Effekte wie eine endlose Scroll-Ansicht oder den endlosen Cover-Flow-Effekt zu erzielen.

Den generischen Typ, um das Datenkarussell zu repräsentieren, erstellen Sie folgendermaßen:

```
class Carousel<T> {
    var items = [T]()
    var currentPosition = 0

    var count: Int {
        return items.count
    }

    var isEmpty: Bool {
        return items.isEmpty
    }

    func append(item: T) {
        items.append(item)
    }

    func next() -> T? {
        if self.isEmpty {
            return nil
        }

        let nextItem = self[currentPosition]
        currentPosition = (currentPosition + 1) % self.count

        return nextItem
    }
```

```
    subscript(position: Int) -> T? {
        if self.isEmpty {
            return nil
        }

        return items[position % self.count]
    }
}
```

Nun können Sie einige `Carousel`-Objekte mit verschiedenen Typen erstellen:

```
var intCarousel = Carousel<Int>()
intCarousel.append(1)

var stringCarousel = Carousel<String>()
stringCarousel.append("Hello")
```

Hinweis

Beachten Sie: Wenn Sie den generischen Typ `Carousel` verwenden, dann müssen Sie den Datentyp angeben, der darin enthalten sein soll.

Sie definieren eine Klasse namens `Carousel`, die mit jedem Typ `T` funktioniert. Dieser `T`-Type braucht kein spezieller Typ zu sein und muss zu keinem Protokoll konform sein. Die Klasse `Carousel` kann mit jedem Typ – von einem einfachen `Int` bis zu einem `UIView` oder anderen selbst definierten Typ – verwendet werden. Im Datenkarussell speichern Sie die Elemente in einem Array vom Typ `T` und die Array-Standardwerte in ein leeres Array. Sie speichern die aktuelle Position im Datenkarussell in einer Eigenschaft namens `currentPosition`. Die Eigenschaft `count` wird an das Array `items` durchgereicht und gibt stets ein `Int` zurück, egal was für ein Typ `T` ist.

Eine andere berechnete Eigenschaft ist `isEmpty`. Damit können Sie schnell testen, ob die Elementzahl gleich `0` ist. Die Methode `append()` akzeptiert den Parameter `item`, der vom Typ `T` sein muss und an das Array `items` angehängt wird. Den wahren Kern dieser Datenstruktur bildet die Methode `next()`. Sie akzeptiert keine Parameter, gibt aber ein optionales `T` zurück. Solange das Datenkarussell mindestens ein Element aufweist, wird das Optional einen Wert enthalten. `nil` ist es nur, wenn das Datenkarussell leer ist. Wenn Sie `next()` aufrufen, wird die Eigenschaft `currentPosition` um eins hochgesetzt und über den Modulo-Operator `%` wird sichergestellt, dass der Wert sich innerhalb des Elementbereichs befindet.

Einen generischen Typ kann man wie jeden anderen Typ auch durch Extensions erweitern und/oder ihn dazu bringen, dass er sich protokollkonform verhält. Als Nächstes sorgen Sie dafür, dass `Carousel` durch eine Erweiterung das Protokoll `Printable` befolgt. Swift definiert das `Printable`-Protokoll folgendermaßen:

Kapitel 7

Die Arbeit mit Objective-C

In diesem Kapitel

Um mit Swift tolle Apps zu erstellen, müssen Sie mit den Frameworks von Cocoa und den Objective-C-Klassen arbeiten. Zum Glück wurde beim Design von Swift auf eine gute Kompatibilität mit Objective-C und allen vorhandenen Frameworks von Apple geachtet, die Sie zum Erstellen von Apps brauchen. In diesem Kapitel lernen Sie sowohl die Grundlagen, um mit Cocoa-Frameworks und Objective-C-Code in Swift zu arbeiten, als auch, wie man den neuen Swift-Code in ein Objective-C-Projekt integriert.

Dies sind die zentralen Themen dieses Kapitels:

- Swift kann Cocoa-Frameworks und Objective-C-Klassen importieren.
- Einige bekannte Objective-C-Klassen werden beim Import automatisch in entsprechende Swift-Klassen konvertiert.
- Einige Swift-Typen werden automatisch in ihre Objective-C-Entsprechungen konvertiert.
- Swift stellt Spezialtypen zur Arbeit mit C-APIs bereit.
- Der Swift-Code kann größtenteils in Objective-C importiert und darin verwendet werden.
- Sie können Swift und Objective-C im gleichen Projekt einsetzen.

7.1 C mit Objective-C-APIs einsetzen

Der erste Schritt bei der Arbeit mit einem Apple-Framework besteht darin, das Modul in den Swift-Code zu importieren. Nach dem Importieren des Frameworks, mit dem Sie arbeiten wollen, haben Sie Zugriff auf alle Klassen und Funktionen dieses Frameworks. Bestimmte Kernklassen werden zudem automatisch den entsprechenden Swift-Klassen zugeordnet. Um ein Objective-C-Framework zu importieren, benötigen Sie nur das Schlüsselwort `import`, gefolgt vom Namen des Frameworks. Ein Import des Cocoa-Frameworks sieht beispielsweise so aus:

```
import Cocoa
```

Nach Import des Frameworks erstellen Sie Swift-Objekte mit Klassen, die im importierten Framework definiert sind. Wenn eine Objective-C-Klasse in Swift importiert ist, laufen verschiedene Konvertierungen automatisch ab. Dabei wird die Syntax fürs Initialisieren einer Objective-C-Klasse so modifiziert, dass sie zur Swift-Konvention passt. Statt `alloc` und `init` aufzurufen, initialisieren Sie die Klassen so wie bei normalen Swift-Klassen, aber die Bezeichnungen der Initialisierer wird geändert, wenn die Klasse in Swift importiert wird. Das Präfix `init` oder `initWith` wird entfernt, und nur die Argumente des Initialisierers verbleiben. Die folgenden Beispiele für die Erstellung von Objective-C-Klassen in Objective-C und Swift veranschaulichen die automatische Umbenennung:

```
//Objective-C:
UIView *view = [[UIView alloc] init];
UIView *viewWithFrame = [[UIView alloc]
➥initWithFrame:CGRectZero];

//Swift:
let view = UIView()
let viewWithFrame = UIView(frame: CGRectZero)
```

Hinweis

Vielleicht ist Ihnen aufgefallen, dass die Methode `alloc` im eben genannten Swift-Code komplett fehlt. Swift wickelt die Speicherzuordnung (*memory allocation*) für Sie ab, auch wenn Sie mit Objective-C-Typen arbeiten. Das bedeutet auch, dass es keine `dealloc`-Methode zum Aufräumen in Ihrem Code gibt. Vielleicht müssen Sie eine `deinit`-Funktion implementieren, damit Sie sicher sind, dass Ihre Klassen ihren Lebenszyklus elegant beenden.

Nun haben Sie initialisierte Objekte und können sie praktisch genauso wie jedes andere Swift-Objekt einsetzen. Sie lesen und schreiben die Eigenschaften genauso wie bei Objective-C:

```
//Objective-C:
view.backgroundColor = [UIColor colorWithWhite:0.5 alpha:0.5];
UIColor *backgroundColor = view.backgroundColor;
```

```
//Swift:
view.backgroundColor = UIColor(white: 0.5, alpha: 0.5)
var backgroundColor = view.backgroundColor
```

Weil `backgroundColor` eine Eigenschaft ist, brauchen Sie kein `()` zu schreiben, um darauf zuzugreifen, so wie Sie es beim Zugriff auf Funktionen machen, die keine Parameter haben. Beachten Sie außerdem, dass die Factory-Methode in Objective-C in die gleiche, für Initialisierer verwendete Syntax konvertiert wurde, damit die Erstellung von Objekten einen konsistenten Stil befolgt.

Wenn Sie einen Wert haben, können Sie Funktionen für das Objekt auf gleiche Weise aufrufen wie bei anderen Swift-Funktionen:

```
//Objective-C:
CGRect frame = CGRectInset(viewWithFrame.frame, 5.0, 5.0);
UIView *subview = [[UIView alloc] initWithFrame:frame];
[viewWithFrame addSubview:subview];

//Swift:
let frame = CGRectInset(viewWithFrame.frame, 5.0, 5.0)
var subview = UIView(frame: frame)
viewWithFrame.addSubview(subview)
```

Objective-C-Typen mit Funktionen, die mehrere Argumente akzeptieren, sehen praktisch genauso aus wie in Objective-C, wenn sie in Swift aufgerufen werden. Der erste Parameter wird ohne externen Namen, aber der Rest der Parameter anhand eines externen Namens übergeben. Dazu ein Beispiel:

```
//Objective-C:
CGRect frame = CGRectInset(viewWithFrame.frame, 5.0, 5.0);
UIView *anotherView = [[UIView alloc] initWithFrame:frame];
[viewWithFrame insertSubview:anotherView aboveSubview:subview];

//Swift:
let frame = CGRectInset(viewWithFrame.frame, 5.0, 5.0)
var anotherView = UIView(frame: frame)
viewWithFrame.insertSubview(anotherView, aboveSubview:subview)
```

7.1.1 Optionale Eigenschaften und Rückgabewerte

In den meisten Fällen verhalten sich Klassen, die aus Objective-C importiert wurden, wie normale Swift-Klassen, aber bei zwei Situationen muss man genau aufpassen: wenn Funktionen Optionals zurückgeben (z.B. `init`-Methoden, die fehlschlagen und `nil` zurückgeben) oder wenn sie Objekte vom Typ `AnyObject` zurückgeben. Objective-C hat keinen optionalen Typ. Daher werden Funktionen, die in Objective-C `nil` zurückgeben können, so modifiziert, dass sie beim Aufruf aus Swift ein Optional zurückgeben. Deshalb muss man mit ihnen in Swift etwas anders umgehen. Schauen wir uns die Methode `viewWithTag` der Methode `UIView` an. Hier haben Sie ein Beispiel dieses Verhaltens und wie es bei Swift

korrekt behandelt wird. Wenn Sie Code schreiben, der nicht prüft, ob ein zurückgegebenes Objekt gültig ist, wird er entweder nicht kompiliert oder stürzt zur Laufzeit ab.

```
//Objective-C:
//Dieser Code produziert zur Laufzeit keinen Absturz
//Bei Objective-C kann man Nachrichten an nil senden
UIView *viewDoesntExist = [viewWithFrame viewWithTag:1];
[viewDoesntExist addSubview:anotherView];

//Swift:
//Dieser Code wird kompiliert, aber abstürzen,
//wenn viewWithTag ein nil zurückgibt
//func viewWithTag(tag: Int) -> UIView?
let viewDoesntExist = viewWithFrame.viewWithTag(1)
//Es ist unsicher, das Entpacken des von einem Objective-C-
// Framework zurückgegebenen Werts zu erzwingen
viewDoesntExist!.addSubview(anotherView)
```

Wenn Sie Code schreiben wollen, der sicher mit Klassen interagiert, die Optionals zurückgeben, müssen Sie die Rückgabewerte auf gleiche Weise prüfen wie bei normalen Optionals:

```
var viewMightExist = viewWithFrame.viewWithTag(1)
if let viewDoesExist = viewMightExist {
    viewDoesExist.addSubview(anotherview)
}
```

Häufig werden Sie Optionals sehen, die als Eigenschaften einer auf Objective-C basierenden Klasse verwendet werden. Schauen wir uns einige Eigenschaften für `UIImageView` an:

```
var image: UIImage? // Default ist nil
var highlightedImage: UIImage? // Default ist nil
```

Beide Eigenschaften sind zur Laufzeit vielleicht nicht vorhanden, und deswegen ist es wichtig, zu prüfen, ob ein gültiger Wert vorhanden ist, wenn Sie auf Eigenschaften zugreifen, die Optionals sind. Sie können per Data Binding eine Variable erstellen und dann im gleichen Schritt das Vorhandensein des Werts überprüfen:

```
var imageView = UIImageView(frame: CGRectZero)
//...
if let image = imageView.image {
    //hier können wir die Eigenschaft image sicher verwenden
}
```

7.1.2 `AnyObject`-Typen

Bei der Arbeit mit Objective-C-Frameworks in Swift müssen Sie auch häufig `AnyObject`-Typen einsetzen. Bei Objective-C gibt eine Funktion üblicherweise einen Wert vom Typ `id` zurück, wenn der Rückgabewert verschiedene Formen annehmen kann; beim Import in Swift wird der Typ `id` in den Typ `AnyObject` konvertiert. Wenn Sie auf einen Wert für ein `AnyObject`-Typ treffen, müssen Sie das Objekt unbedingt auf sichere Weise behandeln, um Abstürze zur Laufzeit zu verhindern.

Eine Funktion, die einen AnyObject?-Typ zurückgibt, ist NSJSONSerialization.JSONObjectWithData(). Um den Rückgabewert dieser Funktion sicher zu verwenden, müssen Sie darauf achten, dass er einen gültigen Wert enthält und außerdem der Wert in den korrekten Typ konvertiert wurde, damit er später im Code verwendet werden kann. Wenn Sie erwarten, dass der Rückgabewert einen bestimmten Typ hat, können Sie Optional Chaining und Type Casting einsetzen. In diesem Beispiel bearbeiten wir den erwarteten Rückgabewert eines NSDictionary:

```
var jsonResult: AnyObject? = NSJSONSerialization.
➥JSONObjectWithData(data, options: nil, error: nil)

if let jsonDict = jsonResult as? NSDictionary {
    //Nun haben wir ein gültiges NSDictionary
}
```

In anderen Situationen müssen Sie vielleicht mit unterschiedlichen Rückgabetypen umgehen können, und mit einer switch-Anweisung prüfen Sie auf sichere Weise verschiedene Resultate. Zuerst prüfen Sie, ob Sie über ein gültiges AnyObject verfügen, und dann suchen Sie über Pattern Matching (Mustererkennung) den Rückgabetyp:

```
var jsonError: NSError?
let jsonOptional: AnyObject! = NSJSONSerialization.
➥JSONObjectWithData(data, options: nil, error: &jsonError)
if let json: AnyObject = jsonOptional {
    switch json {
    case let jsonDict as NSDictionary:
        println(jsonDict)
    case let jsonArray as NSArray:
        println(jsonArray)
    default:
        println("Unknown \(json)")
    }
}
else {
    //bearbeite diesen potenziellen Fehler
}
```

7.1.3 Vererbung, Erweiterungen und Protokolle

Da Sie nun wissen, wie Sie mit Objective-C-Klassen umgehen, die in Swift importiert wurden, wollen wir diese Klassen nach eigenem Bedarf erweitern, und zwar durch Vererbung (Subclassing), Erweiterungen und Protokolle.

Um aus einer Objective-C-Klasse eine Unterklasse zu erstellen, befolgen Sie einfach die Syntax wie bei einer Swift-Klasse. Hier ist ein Beispiel:

```
class MySwiftViewController: UIViewController {

}
```