Swift 3 im Detail

Einführung und Sprachreferenz

Bearbeitet von
Thomas Sillmann

1. Auflage 2017. Buch inkl. Online-Nutzung. 446 S. Softcover
ISBN 978 3 446 45072 1
Format (B x L): 17,7 x 24,4 cm
Gewicht: 922 g

HANSER

Leseprobe

Thomas Sillmann

Swift 3 im Detail

Einführung und Sprachreferenz

ISBN (Buch): 978-3-446-45072-1

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-45072-1

sowie im Buchhandel.

Inhalt

2 Grundlagen der Programmierung

In diesem Kapitel möchte ich Ihnen eine Einführung in die Grundlagen der Programmierung mit Swift geben. Es gibt Ihnen einen ersten Einblick in die Swift Standard Library, zeigt das Erstellen und Verwenden von Variablen und Konstanten und wie Sie Ihren Quellcode mithilfe von Kommentaren dokumentieren.

Wenn Sie dabei sind, Swift zu lernen, empfehle ich Ihnen, die Inhalte der verschiedenen Abschnitte sowie der folgenden Kapitel beispielsweise in einem Playground zu erproben und auszuprobieren, um so möglichst schnell ein Gefühl für die Sprache zu bekommen und selbst aktiv Code zu schreiben.

2.1 Grundlegendes

2.1.1 Swift Standard Library

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

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

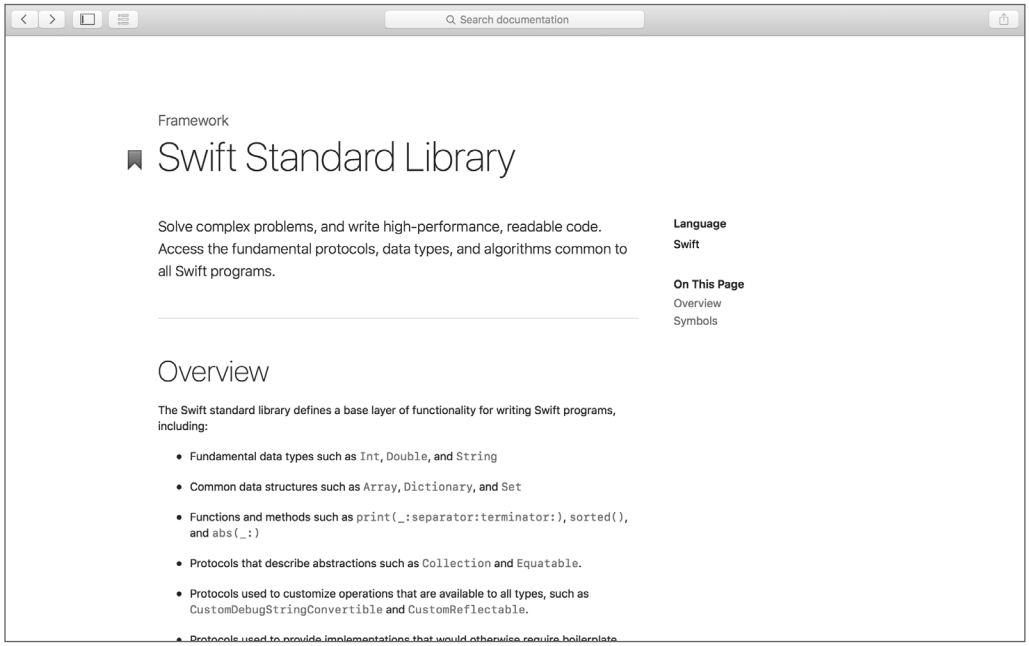

Bild 2.1 Die Swift Standard Library enthält ein umfangreiches Set an Funktionen, die uns bei der Programmierung mit Swift immer zur Verfügung stehen.

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library

Fundamental Type	Beschreibung	Beispiele
`Int`	Ein Integer (`Int`) stellt eine Ganzzahl dar.	`19` `99`
`Float`	Bei `Float` handelt es sich um eine Fließkommazahl	`19.99` `49.94`
`Double`	Auch bei `Double` handelt es sich um eine Fließkommazahl, allerdings ist der Wertebereich von `Double` deutlich größer als der von `Float`; entsprechend belegt ein Double auch mehr Speicherplatz im System als ein `Float`.	`99.19` `94.49`
`Bool`	Bei `Bool` handelt es sich um einen sogenannten Wahrheitswert, dieser kann somit entweder wahr oder falsch (`true` oder `false`) sein.	`true` `false`
`String`	Ein `String` repräsentiert eine Zeichenkette.	`"Mein Name ist Thomas Sillmann."`

Fundamental Type	Beschreibung	Beispiele
`Array`	In einem `Array` können mehrere Werte und Objekte abgelegt werden. Das `Array` erlaubt dann den Zugriff auf die Werte und Objekte, die es hält. Ein `Array` kann dabei beliebige Typen von Werten und Objekten beinhalten.	`["Erster Wert des Arrays", "Zweiter Wert des Arrays"]`
`Dictionary`	Ein `Dictionary` hält mehrere Werte und Objekte ähnlich wie ein `Array`, allerdings ist jeder Wert und jedes Objekt einem einzigartigen Schlüssel innerhalb des Dictionaries zugeordnet. Anhand dieses Schlüssels können dann gezielt Werte ausgelesen, abgefragt und verändert werden.	`["Schlüssel 1": "Wert für Schlüssel 1", "Schlüssel 2": "Wert für Schlüssel 2"]`

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

2.1.2 print

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

Listing 2.1 Einfache Konsolenausgabe mittels `print`

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

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

2.1.3 Befehle und Semikolons

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

Listing 2.2 Schreiben eines Befehls pro Zeile

```
print("Das ist ein erster Befehl.")
print("Anschließend folgt ein zweiter.")
```

```
print("Und zum Abschluss ...")
print("... noch ein vierter!")
```

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

Listing 2.3 Schreiben eines Befehls mit abschließendem optionalen Semikolon

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

Ein Semikolon zum Abschluss ist nur dann Pflicht und zwingend notwendig, wenn Sie *mehrere* Befehle in einer Zeile schreiben möchten (siehe Listing 2.4).

Listing 2.4 Schreiben mehrerer Befehle in einer einzigen Zeile

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

Der letzte Befehl innerhalb der Zeile muss wiederum nicht zwingend mit einem Semikolon abgeschlossen werden, kann es aber optional.

Semikolon – ja oder nein?

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

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

2.1.4 Operatoren

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

Tabelle 2.2 Operatoren in Swift

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

2.2 Variablen und Konstanten

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

2.2.1 Erstellen von Variablen und Konstanten

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

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

Listing 2.5 Erstellen von Variablen und Konstanten

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

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

Listing 2.6 Zugriff auf eine erstellte Variable

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

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

Listing 2.7 Fehler beim Versuch des Änderns einer Konstanten

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

Wann Variable, wann Konstante?

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

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

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

2.2.2 Variablen und Konstanten in der Konsole ausgeben

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

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

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

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

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

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

2.2.3 Type Annotation und Type Inference

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

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

Listing 2.9 Typzuweisung bei der Erstellung von Variablen und Konstanten

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

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

Listing 2.10 Wertzuweisung passend zu den Typen von Variablen und Konstanten

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

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

Listing 2.11 Erstellen neuer Variablen mittels Type Inference

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

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

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

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

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

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

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

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

Zwar ist der gezeigte Code korrekt, allerdings handelt es sich bei `aDouble` nun nicht um eine Variable vom gewünschten Typ `Double`, sondern um eine vom Typ `Int`. Denn Swift vermutet hinter der zugewiesenen Ganzzahl `99` nun einmal keine Fließkommazahl, auch wenn `99` natürlich nichtsdestoweniger ein valider Wert für eine Fließkommazahl wäre. Der Versuch, `aDouble` nun im Nachhinein einen Wert wie `19.99` zuzuweisen, würde ebenfalls in einem Compiler-Fehler enden. Daher ist es in so einem Fall zwingend notwendig, den gewünschten Typ ebenfalls explizit mittels Type Annotation anzugeben, wie in Listing 2.13 zu sehen.

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

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

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

2.2.4 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

Sie haben in Swift die Möglichkeit, sowohl mehrere Variablen und Konstanten direkt in einem Befehl zu erstellen und ihnen dabei auf Wunsch auch bereits Werte zuzuweisen. Dazu beginnen Sie den entsprechenden Befehl entweder mit dem Schlüsselwort `var` (für zu erstellende Variablen) oder `let` (für zu erstellende Konstanten) und benennen dann kommasepariert alle neu zu erstellenden Variablen beziehungsweise Konstanten. Dabei können

Sie entweder allen oder einzelnen Elementen direkt nach dem Namen auf die bekannte Art und Weise einen Wert zuweisen oder einen festen Typ mittels Type Annotation definieren. In Listing 2.14 sehen Sie einige Beispiele dazu, wie dieses Prinzip praktisch angewendet werden kann.

Listing 2.14 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

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

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

2.2.5 Namensrichtlinien

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

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

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

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

Im Zweifel lieber drauf verzichten

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

2.3 Kommentare

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

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

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

Listing 2.16 Ein einzeiliger Kommentar

```
// Ein Kommentar
```

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

Listing 2.17 Ein einzeiliger Kommentar nach einem Befehl

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

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

Listing 2.18 Ein mehrzeiliger Kommentar

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

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

Listing 2.19 Verschachtelte Kommentare

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

Index

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z