Walter Saumweber

SCHNELLEINSTIEG SWIFT 2

160 SEITEN

IN NEUN PRAXISBEZOGENEN SCHRITTEN DIE PROGRAMMIER-SPRACHE FÜR IOS-APPS ERLERNEN.

FRANZIS

Walter Saumweber hat langjährige Erfahrung als Entwickler, Berater und Dozent. Er veröffentlicht regelmäßig Artikel in IT-Fachzeitschriften und ist Autor von zahlreichen Computerfachbüchern. Seine Tätigkeitsschwerpunkte sind die Realisierung von Unternehmenslösungen in Client/Server-Umgebungen und Workflow-Anwendungen.

Walter Saumweber

SCHNELLEINSTIEG SWIFT 2

FRANZIS

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte Daten sind im Internet über http://dnb.ddb.de abrufbar.

Autor: Walter Saumweber
Programmleitung: Dr. Markus Stäuble
Satz: DTP-Satz A. Kugge, München
art & design: www.ideehoch2.de
Druck: CPI-Books
Printed in Germany

ISBN 978-3-645-60256-3

INHALTSVERZEICHNIS

2 SWIFT-BASICS

In diesem Kapitel geht es um die Grundlagen der Programmiersprache Swift. Sie lernen die wichtigsten Syntaxregeln sowie die Verwendung von Konstanten, Variablen und Datentypen kennen.

2.1 Einzeilige und mehrzeilige Kommentare

Lassen Sie uns gleich an den im letzten Kapitel vorgestellten Playground anknüpfen. Die erste Zeile des von Xcode eingefügten Codes lautet dort:

```
//: Playground - noun: a place where people can play
```

Dass dies keine Anweisung darstellt und somit nichts mit dem eigentlichen Swift-Programmcode zu tun hat, liegt auf der Hand. Vielmehr handelt es sich um einen Kommentar – hier mit dem Hinweis, dass es sich bei einem Playground um einen »Spielplatz« handelt. Kommentare werden bei der Übersetzung des Quellcodes ignoriert, sie haben daher keinerlei Einfluss auf das Laufzeitverhalten der fertigen App und sie bringen auch sonst keine Nachteile mit sich. Es handelt sich lediglich um Hinweise, die der App-Entwickler in den Quelltext einfügen kann. Dementsprechend sind die Adressaten von Kommentaren nicht etwa die späteren Benutzer der fertigen App, sondern die Leser des Quellcodes – also der Programmautor selbst oder andere Entwickler, die den Code eventuell zu einem späteren Zeitpunkt überarbeiten. Der gezielte Einsatz von Kommentaren kann ihnen dabei behilflich sein, sich im Code – den sie ja eventuell nicht selbst geschrieben haben – schneller zurechtzufinden. Sie können Kommentare aber z. B. auch als Gedankenstütze einsetzen, etwa wenn Sie beabsichtigen, an einer bestimmten Codestelle zu einem späteren Zeitpunkt noch Verbesserungen durchzuführen.

Swift unterstützt einzeilige und mehrzeilige Kommentare. Einzeilige Kommentare werden mit einem doppelten Slash (`//`) eingeleitet. Der Text, der dem `//` nachfolgt, wird bis zum Zeilenende als Kommentar angesehen. In der oben genannten Kommentarzeile gehört der Doppelpunkt nach den beiden Schrägstrichen folglich nicht zum Kommentarzeichen, sondern zum nachfolgenden Kommentar. Genauso gut hätte man daher auch

```
// Playground - noun: a place where people can play
```

notieren können, und sogar das Leerzeichen nach den beiden Schrägstrichen ist nicht obligatorisch.

Möchten Sie einen ganzen Absatz als Kommentar auszeichnen, brauchen Sie nicht jede Zeile mit einem doppelten Schrägstrich zu beginnen, sondern Sie können die Zeichen `/*` und `*/` verwenden. Das Zeichen `/*` steht für den Anfang, das Zeichen `*/` für das Ende eines Kommentars. Der gesamte Text, der sich zwischen diesen beiden Zeichen befindet, wird als Kommentar angesehen.

```
001 /* Dies ist ein
002 Kommentar, der sich
003 über mehrere
004 Zeilen erstreckt */
```

TIPP:

Apropos »in Kommentar setzen«: Dies ist ein probates Mittel, wenn Sie für einen Testlauf bestimmte Anweisungen beim Kompilieren nicht berücksichtigen wollen. Setzen Sie einfach das Kommentarzeichen `/*` an den Anfang und `*/` an das Ende des auszukommentierenden Codestücks. Der Compiler – das ist die integrierte Software, die den Swift-Quellcode in ausführbaren Code übersetzt – wird die enthaltenen Anweisungen daraufhin ignorieren. Möchten Sie später, dass diese Anweisungen beim Programmlauf wieder ausgeführt werden, entfernen Sie die beiden Kommentarzeichen einfach.

Swift erlaubt es auch, Kommentare zu verschachteln (`/* … /* … */ … */`). Ob dies allerdings überhaupt notwendig oder sinnvoll ist, sei einmal dahingestellt.

2.2 Literale, Rechenoperatoren, Anweisungen

Literale, also konstante Werte, können Sie in Swift, wie bei anderen Programmiersprachen, einfach in den Code schreiben, wobei die üblichen Regeln gelten: Der Nachkommateil von Gleitkommazahlen wird mit Punkt getrennt, für Strings (Zeichenketten) fungieren doppelte Anführungszeichen als Begrenzer.

```
001 99  // ganze Zahl
002 7.5  // Gleitkommazahl
003 "Hello, playground"  // Zeichenkette
```

Zur Darstellung der Grundrechenarten gibt es die aus der Mathematik bekannten Operatoren + für die Addition, – für die Subtraktion, * für die Multiplikation und den Slash (/) für die Division.

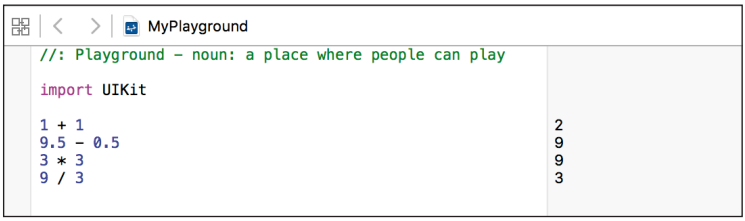


Bild 2.1: Arithmetische Operatoren für die Grundrechenarten

Übrigens handelt es sich bei den im Playground aus Bild 2.1 dargestellten Codezeilen tatsächlich um syntaktisch korrekte Anweisungen, obwohl diese bei realer Ausführung keinerlei Aktionen oder – anders als im Playground – Ausgaben produzieren würden. Swift lässt den Programmierern in dieser Hinsicht genauso wie Objective-C viele Freiheiten. Sogar die Codezeilen

```
001 "Hallo"
002 99
```

mit jeweils nur einem Literal (String sowie ganze Zahl) haben keine Fehlermeldungen oder Warnungen zur Folge, sondern sind syntaktisch gesehen als Anweisungen zu qualifizieren.

Wie Sie sich nach dem Gesagten vermutlich denken können, ist Swift zeilenorientiert. Das heißt, das Ende einer Anweisung geht mit dem Zeilenende einher. Das Setzen eines Semikolons am Ende einer Anweisung ist daher optional. Wenn Sie es aus der Praxis von Objective-C oder anderen Programmiersprachen gewohnt sind, spricht also nichts dagegen, dies zu tun. Es ist auch möglich, mehrere Anweisungen in einer Zeile zu notieren. In diesem Fall müssen Sie die einzelnen Anweisungen – bis auf die letzte – jedoch mit Semikolon abschließen:

```
var x = 12; var y = 7; var z = 1000; print(x + y + z)
```

Die obige Codezeile enthält vier Anweisungen, nämlich drei Variablendefinitionen und eine Ausgabeanweisung. Über deren Bedeutung erfahren Sie im folgenden Abschnitt mehr.

2.3 let und var

Eine Variable ist ein benannter Speicherort, an dem ein Wert abgelegt, gelesen und auch verändert werden kann – »einer«, weil eine Variable jeweils nur einen Wert speichern kann, nicht mehrere gleichzeitig. Von daher hat der Vergleich mit einem Behälter durchaus Sinn.

Wie bei den meisten anderen Programmiersprachen ist auch in Swift einer Variablen immer ein bestimmter Datentyp zugeordnet. Das bedeutet, dass eine einmal definierte Variable grundsätzlich nur Werte eines ganz bestimmten Typs aufnehmen kann (beispielsweise nur Strings oder nur ganze Zahlen).

Swift-Variablen müssen im Code explizit definiert werden, bevor sie verwendet werden können. Dies geschieht mit dem Schlüsselwort `var`. Diesem folgt die Angabe eines Bezeichners, also eines Namens für die Variable.

```
var meineVariable // Definition unvollständig
```

Hier lautet der Name der Variablen `meineVariable`, allerdings ist die Definition noch nicht vollständig. Was noch fehlt, ist die Zuordnung eines Datentyps. Sowohl der Bezeichner einer Variablen als auch deren Datentyp können nach der Definition nicht mehr geändert werden. Es gibt zwei Wege, den Datentyp einer Variablen festzulegen:

- Man notiert einen Doppelpunkt nach dem Variablennamen, gefolgt von einem Leerzeichen und expliziter Angabe des Datentyps.

  ```
  var meineVariable: Int
  var deineVariable: String
  ```

  Die erste Codezeile definiert eine Variable zur Aufnahme von ganzen Zahlen, die zweite Codezeile definiert eine String-Variable. Den Variablen können danach in separaten Anweisungen Werte zugewiesen werden, z. B.:

  ```
  meineVariable = 10
  deineVariable = "Programmieren mit Swift"
  ```

Außerdem unterstützt Swift Typinterferenz. Das heißt, der Compiler kann den Datentyp anhand des zugewiesenen Wertes ermitteln. Sie müssen den Datentyp also nicht zwingend – wie in obigem Beispiel – explizit angeben, wenn Sie der Variablen bereits bei der Definition einen Wert zuweisen.

```
var meineVariable = 10
var deineVariable = "Programmieren mit Swift"
```

Die Variable `meineVariable` ist wieder vom Typ Integer, also zur Aufnahme von ganzen Zahlen bestimmt, und bei der Variablen `deineVariable` handelt es sich nach der obigen Definition wieder um eine String-Variable.

Ausdrücklich sei darauf hingewiesen, dass Folgendes nicht funktioniert:

```
001 var eineVariable // FEHLER
002 // (Es fehlt die Zuordnung eines Datentyps)
003 eineVariable = 17
```

Mit einer separaten Anweisung können Sie die Variable nur initialisieren, wenn Sie bei der Definition das Schlüsselwort für den Datentyp angeben.

```
001 var eineVariable: Int // OK
002 eineVariable = 17
```

Sie können den Datentyp aber auch explizit mit angeben, wenn Sie eine Variable sogleich bei der Definition initialisieren.

```
var eineVariable: Int = 17
```

Das Schlüsselwort für den Datentyp zu notieren kann unter Umständen die Lesbarkeit des Codes erhöhen. Es ist aber in Swift durchaus üblich, die automatische Typerkennung auszunutzen.

EXKURS: DIE ZUWEISUNG

Lassen Sie uns an dieser Stelle für die in der Programmierung weniger versierten Leser kurz auf die Zuweisung eingehen. Dabei handelt es sich um eine spezielle Anweisung, mit deren Ausführung in einer Variablen ein neuer Wert abgelegt wird. Der alte Wert wird dabei gegebenenfalls überschrieben, da eine Variable zur gleichen Zeit ja nur einen einzigen Wert enthalten kann. Bei der ersten Zuweisung an eine Variable spricht man von Initialisierung: Die Variable wird mit einem Wert initialisiert.

Die Zuweisung wird mit dem aus der Mathematik bekannten Gleichheitszeichen (dem Zuweisungsoperator) dargestellt, das hierbei aber eine ganz andere Bedeutung hat. Eine Zuweisung setzt sich insgesamt aus drei Teilen zusammen:

- dem Zuweisungsoperator (=)
- einem linken Operanden; er besteht immer aus genau einem Variablennamen, wobei man vom Bezeichner der Variablen spricht. Damit wird die Variable bezeichnet, die mit der Zuweisung einen neuen Wert erhält.
- dem rechten Operanden, der den zuzuweisenden Wert darstellt. Hier darf auch ein komplexer Ausdruck stehen, der nach seiner Berechnung den zuzuweisenden Wert ergibt.

Hier ein paar Beispiele:

```
var zahl = 33 // Definition der Variablen zahl
zahl = 22
```

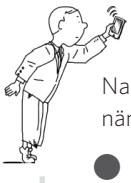
Die Variable `zahl` wird gleich bei der Definition mit dem Anfangswert 33 versehen (initialisiert). Nach der Zuweisung `x = 22` besitzt die Variable den Wert 22.

```
zahl = 8 * 3 - 11
```

Nach Ausführung dieser Anweisung enthält die Variable `zahl` den Wert 13. Daraus wird auch deutlich, dass eine Zuweisung immer von rechts nach links abgearbeitet wird:

1. Zunächst wird der Ausdruck auf der rechten Seite ausgewertet.
2. Anschließend wird das Ergebnis der auf der linken Seite bezeichneten Variablen zugewiesen.

```
zahl = zahl + 7
```



Nach Ausführung der obigen Anweisung enthält `zahl` nunmehr den Wert 20, nämlich 13 plus 7. Es gilt:

- Wenn der Bezeichner einer Variablen auf der rechten Seite einer Zuweisung steht, wird der aktuelle Wert der Variablen gelesen.
- Auf der linken Seite der Zuweisung bezeichnet der Variablenname die Speicherstelle, »die etwas bekommt«, nämlich den Wert, der sich nach Auswertung des Ausdrucks auf der rechten Seite ergibt.

```
zahl = "Swift" // FEHLER
```

Die obige Anweisung ist fehlerhaft, da im Zuge der Definition durch die Zuweisung des Wertes 33 an die Variable `zahl` für diese per Typinterferenz der Datentyp Integer festgelegt wurde und einer Integer-Variablen kein String zugewiesen werden kann.

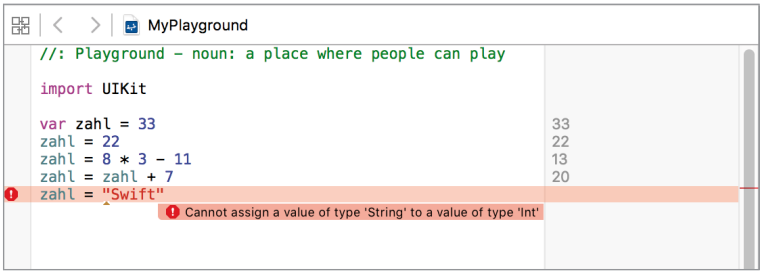


Bild 2.2: Neben fehlerhaften Codezeilen erscheint im Playground ein Symbol mit weißem Ausrufezeichen auf rotem Hintergrund. Klicken Sie das Symbol gegebenenfalls an, um die komplette Fehlermeldung zu sehen.

Eine App arbeitet häufig mit unveränderlichen Werten, wie z. B. dem aktuellen Mehrwertsteuersatz, der maximalen Anzahl von Login-Versuchen, die Sie als Programmautor erlauben wollen, oder bei Spielen mit einer maximal zu erreichenden Punktzahl. Dabei ist es wenig praktikabel, die entsprechenden Werte als Literale in den Code zu schreiben, vor allem dann nicht, wenn sie öfter vorkommen. Wenn sich z. B. der Mehrwertsteuersatz ändert, müssten Sie den Wert an jeder Stelle, an der er auftritt, austauschen. Verwenden Sie dagegen Konstanten, müssen Sie den Code nur an einer einzigen Stelle ändern. Außerdem erhöht sich die Lesbarkeit des Quellcodes, da eine Konstante wie `mwst`, `anzahlErlaubterLogins` oder `maxPunkte` natürlich aussagekräftiger ist als z. B. die Literale `19`, `3` oder `100` (falls man

drei Login-Versuche erlauben möchte bzw. in einem Spiel die maximal zu erreichende Punktzahl 100 beträgt).

Wie bei Variablen handelt es sich auch bei Konstanten um einen mit Namen ansprechbaren Speicherort – von daher spricht man auch von »benannten Konstanten«, um sie von Literalen zu unterscheiden –, und Konstanten müssen ebenfalls vor ihrer Verwendung definiert werden. Dies geschieht mit dem Schlüsselwort `let`.

```
let maximum = 100
```

Während Variablen ihren Wert durch Neuzuweisung ändern können, ist das bei Konstanten – wie der Name schon sagt – nicht der Fall.

```
001 let maximum = 100
002 maximum = 150  // FEHLER
```

Die obige Neuzuweisung ist fehlerhaft, da `maximum` als Konstante definiert wurde. Die Verwendung von Konstanten für Werte, die während der App-Ausführung unveränderlich bleiben sollen, dient somit dem Schutz des Programmierers vor eigenen Fehlern. Gegenüber Variablen haben Konstanten zudem den Vorteil, dass sie effizienter verwaltet werden (der Compiler »weiß«, dass sich der Wert nicht mehr ändern wird) und somit ressourcensparender sind.

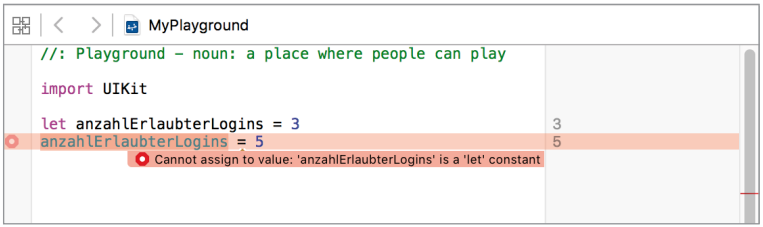


Bild 2.3: Die erneute Zuweisung eines Wertes an eine Konstante macht sich auch im Playground als Fehler bemerkbar.

INFO:

Sie müssen den Wert einer Konstanten jedoch nicht zwangsläufig bei der Definition zuweisen, sondern können dies wie bei Variablen auch in einer separaten Anweisung tun.

```
let anzahlErlaubterLogins: Int
anzahlErlaubterLogins = 3  // OK
```

Nach der ersten Zuweisung dürfen Sie jedoch keine weitere Zuweisung durchführen.

Wenn Sie mehrere Variablen oder Konstanten benötigen, können Sie sie auch mit einer einzigen Anweisung definieren, indem Sie die einzelnen Ausdrücke durch Kommas trennen:

```
var zahl1 = 0, zahl2 = 0, zahl3 = 0
```

oder

```
var zahl1: Int, zahl2: Int, zahl3: Int
```

Die erste Anweisung definiert die Integer-Variablen `zahl1`, `zahl2`, `zahl3` und initialisiert sie mit dem Wert `0`. Die zweite Anweisung definiert die Variablen ebenfalls mit dem Datentyp Integer, ohne sie jedoch zu initialisieren. Die beiden Anweisungen sind natürlich alternativ zu verstehen, da die Bezeichner von Variablen eindeutig sein müssen. Wenn Sie Konstanten benötigen, verwenden Sie das Schlüsselwort `let`:

```
let anzahlErlaubterLogins = 3, maxPunkte = 100
```

Die obigen Beispiele definieren durchweg Variablen bzw. Konstanten gleichen Datentyps. Eine Einschränkung in dieser Hinsicht besteht jedoch nicht. Sie können in einer Codezeile auch Variablen bzw. Konstanten unterschiedlichen Datentyps definieren:

```
var anzahlLogins = 0, str = "Welcome to Swift"
```

2.4 Bezeichnerwahl

Für verschiedene Elemente der Programmiersprache, wie z. B. Variablen, Konstanten, Klassen oder Funktionen, vergeben Sie Namen, wenn Sie sie im Code definieren. Der Fachausdruck für solche Namen lautet Bezeichner.

Grundsätzlich dürfen Sie Bezeichner unter Beachtung einiger Regeln frei wählen:

- Das erste Zeichen darf keine Ziffer sein.
- Swift ist case-sensitiv, das heißt, es unterscheidet zwischen Groß- und Kleinschreibung. Die Bezeichner `swiftIsCool` und `swiftiscool` sind daher nicht identisch, sondern bezeichnen im Code zwei unterschiedliche Elemente.
- Bezeichner müssen eindeutig sein. Dies hat zur Folge, dass innerhalb eines Gültigkeitsbereichs nicht der gleiche Bezeichner für zwei verschiedene Elemente vergeben werden darf.
- Ebenso wenig dürfen Bezeichner mit Schlüsselwörtern gleichlauten, die ja bereits eine in der Programmiersprache Swift festgelegte Bedeutung haben. Es gibt jedoch eine Ausnahme: Wenn Sie das Schlüsselwort mit den Zeichen `` `...` `` (oder ´...´) begrenzen, können Sie den Namen als Bezeichner verwenden. Der Aufwand erscheint allerdings unbegründet, und eine Notwendigkeit, dies zu tun, ist auch nicht ersichtlich.

Ansonsten dürfen Swift-Bezeichner nahezu alle Zeichen des Unicode-Zeichensatzes enthalten, also auch deutsche Umlaute und sogar chinesische, japanische, griechische Schriftzeichen etc. Für die Kreiszahl Pi könnten Sie z. B. auch eine Konstante mit dem Bezeichner π definieren. Einige Zeichen sind jedoch in Bezeichnern nicht erlaubt. Dazu gehören, wie in jeder anderen Programmiersprache, Leerzeichen (der Compiler könnte ansonsten nicht ermitteln, wo ein Bezeichner endet), aber auch verschiedene Sonderzeichen wie z. B. mathematische Symbole, Punkte, Pfeile oder Rahmenzeichen. Am besten fahren Sie, wenn Sie auf Sonderzeichen gänzlich verzichten. Unterstriche in Bezeichnern sind zwar erlaubt, jedoch in der Swift-Programmierung nicht üblich.

UNICODE

Der Unicode-Zeichensatz ermöglicht die Darstellung von bis zu 65536 verschiedenen Zeichen. Er enthält alle Sonderzeichen und sogar jedes Schriftzeichen jeder beliebigen Landessprache. Jedem Zeichen ist eine Ordnungszahl eindeutig zugeordnet. Die Ordnungsnummern sind fortlaufend. Die ersten 256 Zeichen entsprechen dabei der früher durchweg verwendeten ASCII-Tabelle.

Für die Bezeichner von Variablen und Konstanten wird in Swift die lowerCamelCase-Schreibweise empfohlen. Nach dieser beginnt jeder Bezeichner mit einem Kleinbuchstaben. Setzt sich ein Bezeichner aus mehreren Wörtern zusammen, dann wird jedes weitere Wort großgeschrieben. Hier ein paar Beispiele: `meinMacBook`, `bigPoint`, `teilnehmer` (setzt sich nicht aus mehreren Wörtern zusammen), `anzahlTeilnehmer`, `arrayOfSelectedItems`, `minValue`, `maxValue`.

Hier ein paar weitere Beispiele für gültige Bezeichner:

```
001 var kmProSek = 0
002 let π = 3.14159265359
003 var □□ = "biànliàng"
004 var myVar1 = 99
005 var myVar2 = 5000
```

Folgende Bezeichner sind dagegen nicht korrekt:

```
001 var my Var: Int  // keine Leerzeichen in Bezeichnern
002 let 1A = 100  // das erste Zeichen eines Bezeichners
003               // darf keine Ziffer sein
004 var Int = "Integer"  // Int ist ein Schlüsselwort zur
005                      // Bezeichnung des Datentyps
006                      // und darf deshalb nicht zur
007                      // Bezeichnerwahl herangezogen werden
```

Und eine weitere Empfehlung: Grundsätzlich können Sie für die Bezeichner von Variablen oder Konstanten auch nur einen einzigen Buchstaben vorsehen, z. B. `x`, `y` oder `z`, und in einigen Fällen ist das auch praktikabel – etwa bei einer Schleifenvariablen, die ansonsten keine spezielle Bedeutung hat. Im Allgemeinen sollten Sie jedoch »sprechende Namen« bevorzugen. Das macht den Quellcode besser lesbar. Bei einer Variablen `skonto` weiß man beispielsweise sofort, was gemeint ist, während die Bedeutung einer Variablen `x` oder `s` (für Skonto) allenfalls aus dem Zusammenhang erkennbar ist.

Ein ganz besonderes Gimmick der Programmiersprache Swift sei in diesem Zusammenhang noch erwähnt: Swift erlaubt es, als Bezeichner sogenannte Emoji-Symbole zu verwenden. Emoji-Symbole stammen ursprünglich aus Japan. Sie werden dort seit längerem vergleichbar den bei uns geläufigen Emoticons eingesetzt.

INFO:

Emoticons sind Symbole, die Stimmungen wiedergeben. Die bei uns am meisten verwendeten Emoticons sind

`:-)`
`:-(`
und
`;-)`

Zwar haben die japanischen Mobilfunkbetreiber eigene Standards in Bezug auf Tastatur und Zeichenbelegung. Infolge der häufigen Verwendung und auch auf Betreiben von Apple und Google entschied sich das Unicode-Konsortium jedoch, 600 Emoji-Zeichen in den Unicode-Zeichensatz aufzunehmen. Ob sich der Aufwand bei der Programmierung für Sie lohnt, sollten Sie für sich selbst entscheiden.

```
let 🐶 = "Mein Hund"
print(🐶)
```

Bild 2.4: Emoji-Zeichen als Bezeichner einer Konstanten

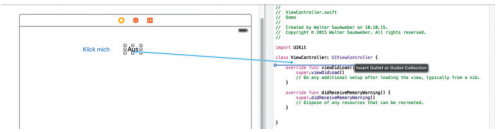
TIPP:

Emojis sind als Nicht-Buchstabenzeichen mit den deutschen Tastaturen auf einfache Weise nicht darstellbar. Xcode stellt jedoch eine Vielzahl von Emojis, geordnet nach Kategorien (*Personen*, *Natur*, *Essen & Trinken*, *Feier*, *Aktivität*, *Reisen & Orte*, *Objekte & Symbole* usw.), im Menü zur Verfügung. Wählen Sie in der oberen Menüleiste *Edit/Emoji & Symbols*. Im erscheinenden Fenster brauchen Sie ein Emoji nur auszuwählen, um es im Code zu verwenden. Es wird dann an der aktuellen Cursorposition eingefügt. Per Eingabe in das Suchfeld können Sie auch gezielt nach bestimmten Emojis suchen. Beachten Sie jedoch, dass Sie englische Suchbegriffe verwenden müssen. Suchen Sie beispielsweise nicht nach »Hund«, sondern nach »dog«.

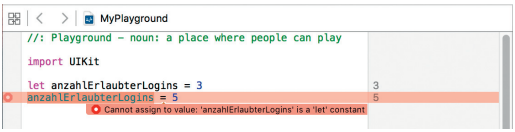
STICHWORTVERZEICHNIS

SCHNELLEINSTIEG SWIFT 2

Egal ob Sie bereits für Apple iOS Apps entwickeln oder damit anfangen wollen: um die Programmiersprache Swift kommen Sie nicht herum. Apple hat alles über Bord geworfen und von Grund auf eine neue Sprache entwickelt. Endlich, werden viele Objective-C-Entwickler sagen. Und Neulinge brauchen sich nicht mehr mit den Altlasten auseinanderzusetzen. Nun liegt bereits die zweite Generation der Sprache Swift vor, und Entwickler müssen sich mit den neuen Konzepten auseinandersetzen. Dieses Buch zeigt Ihnen, worauf es dabei ankommt.



Die Nutzung der Entwicklungsumgebung Xcode wird anhand von Bildern erklärt.



Typische Fehler während der Programmierung werden erklärt, damit Sie sie vermeiden.

In neun Schritten zum Erfolg

Nach der Installation der Entwicklungsumgebung Xcode steigt der Autor direkt in die Swift-Programmierung ein, angefangen bei der grundlegenden Syntax wie den Kommentaren, Literalen, Rechenoperatoren, Anweisungen und Datentypen bis zu den Kontrollstrukturen. Damit können Sie direkt erste Programme mit Swift umsetzen. Für die ersten Gehversuche müssen Sie kein großes Xcode-Projekt beginnen, sondern können sich mit dem Playground in der IDE begnügen.

Danach geht es direkt weiter mit Funktionen, Funktionstypen und Closure-Ausdrücken. Die letzten beiden Kapitel im Buch beschäftigen sich mit der objektorientierten Programmierung. Sie lernen dabei auch OOP-Konzepte wie Lazy und Computed Properties kennen.

Aus dem Inhalt:

9 783645 602563

Besuchen Sie unsere Website
www.franzis.de

FRANZIS