

---

## 6 Fehlersuche und Problembehandlung

Die Grundlagen der App-Entwicklung kennen Sie nun. Aber so sorgfältig man auch arbeitet, hin und wieder schleicht sich der eine oder andere Fehler ein. In diesem Kapitel erfahren Sie, wie Sie Fehler finden und (vielleicht) auch vermeiden können.

### 6.1 Breakpoints im Quellcode setzen

Einen großen Teil der Zeit bei der App-Entwicklung verbringt man wohl damit, Code zu schreiben, in den sich jedoch der eine oder andere Fehler einschleichen kann. Xcode stellt zum Glück Werkzeuge bereit, um Fehler im Code zu finden. Oft werden Sie nämlich nicht wissen, an welcher Stelle im Quellcode ein Fehler auftritt. Aus diesem Grund muss man sich an die entsprechenden Stellen erst einmal heranpirschen. Möglich wird das mit dem in Xcode integrierten Quellcode-Debugger.

Der Debugger ermöglicht es, den Programmfluss an einer markierten Stelle zu unterbrechen und von dieser Stelle aus Variablen zu inspizieren oder auch die Programmausführung schrittweise fortzusetzen. Der Debugger von Xcode wird auf sehr simple Weise aktiviert. Hierzu müssen Sie im Codefenster zunächst lediglich an der gewünschten Stelle einmal den Mausbutton betätigen. Anschließend wird an dieser Stelle eine Markierung (Breakpoint) gesetzt und die entsprechende Codezeile farblich hervorgehoben (siehe Abb. 6–1). Natürlich können auch mehrere Haltepunkte an unterschiedlichen Stellen gesetzt werden.

Startet man die Anwendung anschließend über Xcode, so wird das Programm genau bis zum zuvor markierten Haltepunkt ausgeführt. Die weitere Ausführung des Programms wird nun angehalten und befindet sich quasi in einem Pausenmodus. An dieser Stelle kommen dann die Funktionen des Debuggers zur Steuerung des weiteren Programmflusses zum Tragen.

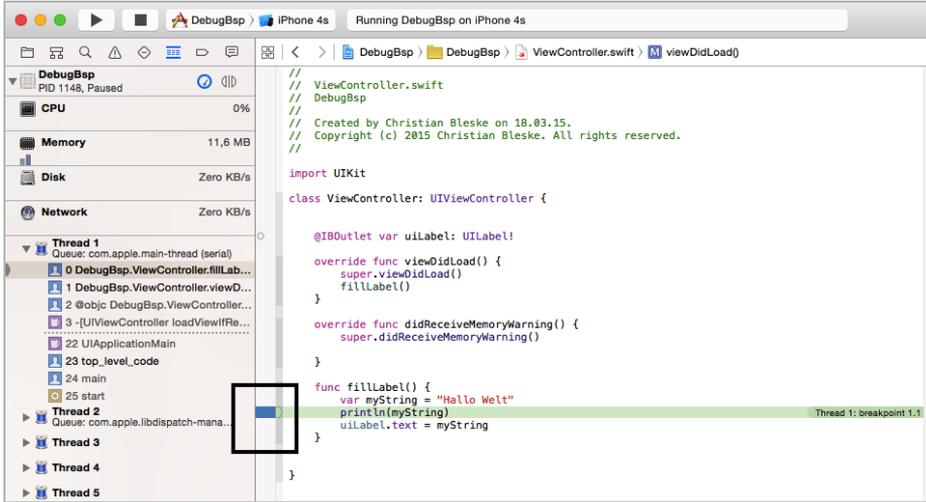


Abb. 6–1 Einen Breakpoint im Quellcodefenster einfügen

**Hinweis**

Um einen Breakpoint wieder zu entfernen, müssen Sie mit einem Rechtsklick das Kontextmenü oberhalb des Breakpoints aufrufen. Im Menü gibt es dann den Punkt *Delete Breakpoint*. Mit der Funktion *Disable Breakpoint* lässt sich ein Haltepunkt vorübergehend deaktivieren.

Direkt unterhalb des Quellcodefensters befindet sich eine Leiste mit unterschiedlichen Schaltflächen, die es erlauben, den Programmfluss unter unterschiedlichen Bedingungen fortzusetzen.

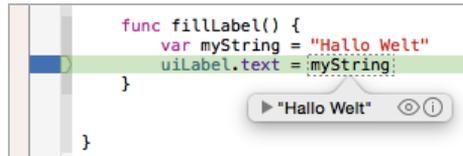


Abb. 6–2 Optionen zur Codeausführung im Debugger

Die erste Schaltfläche (*Toogle global breakpoint state*; siehe Abb. 6–2 von links nach rechts) ermöglicht es Ihnen, an zentraler Stelle den bzw. die gesetzten Haltepunkte ab- und auch wieder einzuschalten. Soll die Programmausführung fortgesetzt werden, so muss die folgende Schaltfläche mit dem Dreieckssymbol (*Continue program execution*) betätigt werden. Die dritte Schaltfläche ist der *Step Over* Button. Die Betätigung des Buttons ermöglicht es, eine folgende Anweisung, z.B. einen Funktionsaufruf, zu überspringen. Die beiden folgenden Schaltflächen (*Step into* bzw. *Step out*) haben eine ähnliche Funktion. Mit *Step into* kann in eine Funktion und mit *Step out* aus einer Funktion heraus »gesprungen« werden.

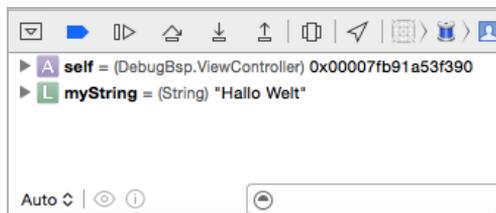
## 6.2 Inspizieren von Variablen

Durch die Verwendung von Haltepunkten haben Sie die Möglichkeit, sich die Inhalte von Variablen zu einem bestimmten Zeitpunkt einmal näher anzusehen. Hierfür gibt es unterschiedliche Möglichkeiten. Zum einen wäre da die Methode, den Mauszeiger oberhalb der zu inspizierenden Variablen zu platzieren.



**Abb. 6-3** Inhalt von Variablen ansehen

Sobald der Mauszeiger zur Laufzeit oberhalb einer Variablen positioniert wird, wird der Inhalt der Variablen in einem kleinen Fenster automatisch angezeigt. Die beiden Symbole im Fenster (das *Auge* und das *I*) geben zum einen Auskunft über den Typ, und zum anderen wird die Zuweisung noch einmal detailliert angezeigt. Zeitgleich wird der Inhalt der Variablen auch noch im *Variables View* angezeigt, das sich direkt unterhalb des Quellcodefensters befindet.



**Abb. 6-4** Inhalt von Variablen im Variables-View

Eine weitere Möglichkeit, sich den Inhalt von Variablen anzusehen, haben Sie bereits kennengelernt. Mit der *print*-Anweisung können Inhalte von Variablen im *Output-Fenster* von Xcode angezeigt werden.



**Abb. 6-5** Inhalt einer Variablen im Output-View

Zur Anzeige ist es natürlich notwendig, dass man zuvor die *print*-Anweisung im Quellcode auch einsetzt:

```
func fillLabel() {
    var myString = "Hallo Welt"
    print(myString)
    UILabel.text = myString
}
```

Sobald nun die *print*-Anweisung ausgeführt worden ist, wird im *Output Window* von Xcode die entsprechende Information ausgegeben.

### Hinweis – Bedingungen für Haltepunkte

Manchmal möchte man den Inhalt einer Variablen zu einem bestimmten Zeitpunkt sehen. Wie lässt sich das bewerkstelligen? Hierzu gibt es die Möglichkeit, für einen Breakpoint Bedingungen festzulegen, die zutreffen müssen, damit die Programmausführung angehalten wird.



Abb. 6–6 Bedingung für einen Haltepunkt festlegen

Damit eine Bedingung für einen Haltepunkt festgelegt werden kann, müssen Sie ihn setzen und anschließend im Kontextmenü zum Breakpoint den Menüpunkt *Edit Breakpoint...* aufrufen. Anschließend öffnet sich ein kleiner Dialog, den Sie in Abbildung 6–6 sehen. Das Beispiel besteht aus einer kleinen Schleife:

```
func doLoop() {
    var myvar:String
    var index = 0

    while (index < 5) {
        myvar = String(index)
        print(myvar)
        index += 1
    }
}
```

Diese Schleife soll durch Setzen eines Haltepunkts untersucht werden, aber nicht nach dem ersten, sondern erst nach dem dritten Durchlauf. Hierzu wird innerhalb des *Condition*-Feldes nun eine Bedingung erfasst (sie muss einen booleschen Ausdruck ergeben). Im Beispiel wurde die Bedingung *index==3* (Achtung: C-Notation!) erfasst. Wird die Anwendung nun gestartet, so wird die Programmausführung angehalten, sobald der entsprechende Wert erreicht ist.

### print bzw. Print vs NSLog

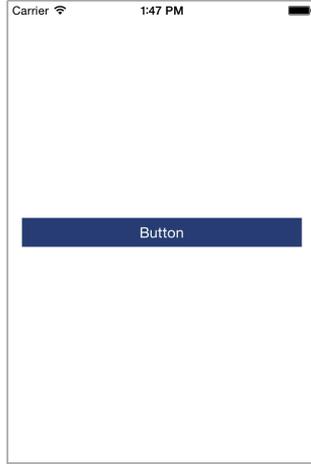
Neben der *print*-Anweisung gibt es noch eine weitere Anweisung, um Informationen im Output-Fenster auszugeben. Alte Apple-Entwickler kennen diese Anweisung bereits: *NSLog*. Hierbei handelt es sich um eine Anweisung, die bisher nur in Objective-C verwendet wurde. *NSLog* ist somit (quasi) das Gegenstück zu *print* für Objective-C. *NSLog* kann aber natürlich auch in Swift verwendet werden. Hierbei sollten Sie aber beachten, dass es einige Unterschiede gibt:

- *NSLog* ist langsamer in der Verarbeitung als *print*.
- Neben dem eigentlichen Wert übergibt *NSLog* noch einen Zeitstempel.
- *NSLog* synchronisiert die Ausgabe bei der Verwendung von Threads. Mit *print* ist das nicht der Fall. Hier kann es also zu einem Durcheinander bei der Ausgabe kommen.
- Bleibt *NSLog* im Release-Code enthalten, so verlangsamt es das Laufzeitverhalten einer Anwendung, da es in der Konsole des jeweiligen Geräts ausgegeben wird. Informationen, die mit *print* ausgegeben werden, werden nur in der Debug-Konsole von Xcode ausgegeben.

## 6.3 View Debugging

Nicht nur Quellcode lässt sich ab Xcode 6 debuggen, sondern es gibt sogar ein neues Werkzeug innerhalb der Entwicklungsumgebung, mit dem sich Views zur Laufzeit der Anwendung untersuchen und Fehler innerhalb der Darstellung finden lassen. Sehen Sie sich hierzu einmal die Beispiel-App des Kapitels an. Diese enthält neben einem Label noch zusätzlich einen Button. Beide Elemente sind mittig platziert. Das bedeutet, zur Laufzeit wird das Label-Control vom Button verdeckt. Startet man die Anwendung, so ist nur der Button zu sehen. Warum aber das Label nicht angezeigt wird, ist auf den ersten Blick nicht zu erkennen (siehe Abb. 6–7).

Nach dem Start der App sollte nun zunächst wieder Xcode aktiviert werden, ohne dabei allerdings die laufende App zu beenden. Öffnen Sie die View-Ansicht, indem Sie im Project Navigator die Datei *Main.Storyboard* markieren. Direkt unterhalb des Interface Builders wird eine Leiste mit mehreren Symbolen angezeigt. Betätigen Sie den Button *Debug View Hierarchy*, der aus zwei übereinandergelegten Rechtecken besteht (siehe Abb. 6–8).

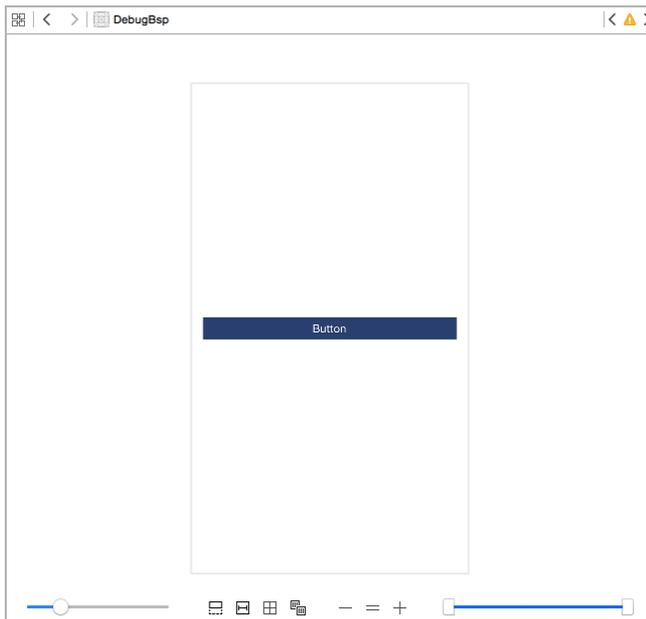


**Abb. 6-7** Die App zur Laufzeit



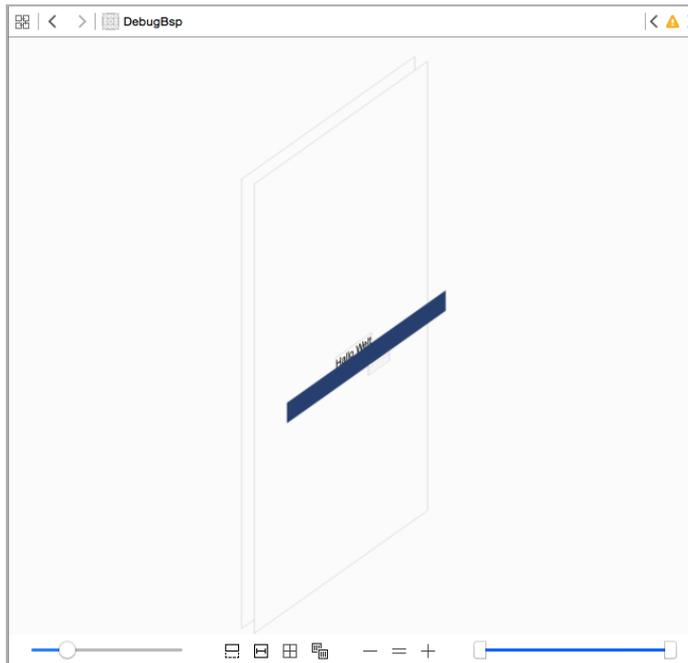
**Abb. 6-8** Die App zur Laufzeit

Es dauert nur einen Moment; anschließend ist der Debugging-Modus für Views aktiviert:



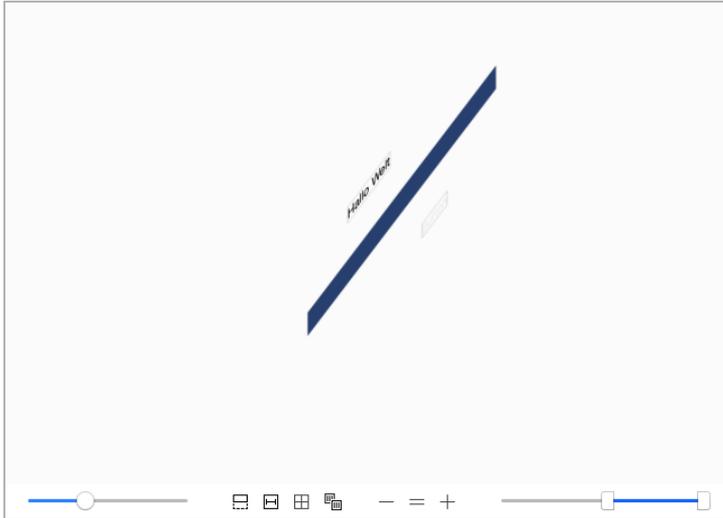
**Abb. 6-9** Debug-View-Modus

Auf den ersten Blick unterscheidet sich der Debug-Modus für Views nicht von der normalen Design-Ansicht. Erst auf den zweiten Blick fallen zusätzliche Controls auf. Mit diesen Controls und unter Verwendung der Maus lässt sich das View nämlich aus unterschiedlichen Positionen heraus betrachten. Klicken Sie mit der Maus in die View-Ansicht, und halten Sie die Maustaste gedrückt. Wenn Sie nun die Maus bewegen, so wird sich auch das View bewegen.



**Abb. 6-10** View, nachdem es gedreht wurde

Ihnen wird sofort auffallen, dass – sobald das View in die eine oder andere Richtung gedreht worden ist – die Ebenen (Label, Button und View) nicht mehr direkt übereinanderliegend angezeigt werden, sondern mit etwas Abstand zwischen den Controls. Dieser Abstand ist vorhanden, damit man die einzelnen Elemente besser auseinanderhalten kann. Der Abstand kann sogar noch vergrößert oder auch verkleinert werden. Hierzu wird der Schieberegler verwendet, der sich in der linken unteren Ecke befindet. Auch in der rechten Ecke befindet sich ein Schieberegler, der von beiden Seiten aus verwendet werden kann. Mit ihm lassen sich die Ebenen stufenweise ein- und ausblenden.



**Abb. 6-11** Ebenen lassen sich ein- und ausblenden.

Diese Funktion ist natürlich nicht zum Selbstzweck vorhanden. Mit ihr lässt sich visuell relativ einfach ermitteln, wieso in einem View etwas gerade nicht angezeigt wird.

Im Beispiel ist so gut zu erkennen, dass das Label- vom Button-Control verdeckt wird. Zur Rechten befinden sich drei Schaltflächen mit den Symbolen -, = und +. Sie werden benutzt, um in und aus dem View heraus- und hereinzuzoomen sowie um die ursprüngliche Größe wiederherzustellen. Jedes Element im View kann mit der Maus markiert werden.

Auch die im View verwendeten Constraints lassen sich anzeigen. Hierzu wird die zweite Schaltfläche von links verwendet. Wenn Sie sie anklicken, werden die im View vorhandenen Constraints angezeigt. Mit der letzten Schaltfläche (*Adjust the view mode*) lassen sich Elemente ein- und ausblenden. So kann man über eine Option beispielsweise festlegen, dass nur der Rahmen der Elemente (*Wireframes*) oder lediglich der Inhalt (*Contents*) der Controls angezeigt wird.

## 6.4 Fehlerbehandlung mit »try catch«, (NSError & Co.

In Anwendungen können Fehler auftreten – sei es, weil Eingaben nicht verarbeitet werden können oder weil beispielsweise ein entferntes System, das angesprochen werden soll, nicht zur Verfügung steht. In solchen Situationen sollte das Programm natürlich nicht sofort abstürzen, sondern den Anwender darüber informieren, dass ein Problem vorliegt, und ggf. Alternativen anbieten.

In diesen Situationen kommt neben der Klasse *NSError* seit der Version 2.0 von Swift auch ein *do...try...catch*-Block zum Einsatz. Hierbei wird Code, bei dessen Ausführung möglicherweise ein Fehler auftritt, in einen Sicherheitsblock

gesteckt. Sofern eine Aktion nicht das gewünschte Resultat erbringt und einen Fehler auslöst, kann innerhalb eines *catch*-Blocks über eine *NSError*-Instanz der Fehlerursache auf den Grund gegangen werden. Das Arbeiten mit *do...try...catch* und einem *NSError*-Objekt ist zum Glück relativ einfach. Die *NSError*-Instanz enthält üblicherweise eine Fehlermeldung, wenn ein Problem vorliegt. Wenn kein Problem aufgetreten ist, enthält sie einen *nil*-Wert, und der *catch*-Abschnitt wird übersprungen.

Die beschriebene Vorgehensweise erinnert dabei sehr an den Umgang mit *Exceptions* in *Java* oder *C#*. Im einfachsten Fall wird versucht, eine Funktion auszuführen. Sollte innerhalb der Funktion ein Fehler auftreten, so kann das *NSError*-Objekt untersucht und eine Fehlermeldung sowie ein Fehlercode ausgelesen werden. Das könnte beispielsweise wie folgt aussehen:

```
func errorSample() {
  let jstring = "{ \"test\": \"bsp\" }"
  let daten = jstring.data(using: String.Encoding.utf8, allowLossyConversion:
    true)
  var nsError = NSError()

  var jresult : Any?
  do {
    jresult = try JSONSerialization.jsonObject(with: daten!, options:
      .allowFragments)
  } catch let error as NSError {
    nsError = error
    jresult = nil
  }

  if jresult != nil {
    print("Alles ok!")
  } else {
    print("Fehler bei Umwandlung")
  }
}
```

Im Beispiel sollen Daten im JSON-Format verarbeitet werden, die in einer Variablen vom Typ *String* gespeichert sind. Wenn Sie JSON noch nicht kennen, finden Sie in Kapitel 9 mehr Informationen zu diesem Thema. Im Moment nehmen Sie einfach hin, dass Daten im JSON-Format zum Datenaustausch verwendet werden.

Die ersten beiden Zeilen im Beispiel dienen der Initialisierung der Variablen. Der Variablen *jstring* wird eine Zeichenkette zugewiesen, die im JSON-Format formatiert ist. Mit der Methode *data* wird die Zeichenkette UTF8 encodiert. Das eigentliche Beispiel beginnt mit der Definition des *do...try...catch*-Blocks und des *NSError*-Objekts *nsError*. Da nicht bekannt ist, ob ein Fehler auftritt, wird das Objekt als optional deklariert. Der Block zur Fehlerbehandlung wird mithilfe des Schlüsselwortes *do* eingeleitet.

In der folgenden Zeile wird mit der Methode *JSONObjectWithData* der Klasse *NSJSONSerialization* aus den Daten im JSON-Format ein in Swift lesbares Objekt erstellt. Beachten Sie hier bitte, dass der Methode *JSONObjectWithData* nicht nur die Daten im JSON-Format als Parameter übergeben werden, sondern dass auch das Schlüsselwort *try* verwendet wird, um zu kennzeichnen, dass hier auch ein Fehler auftreten kann. Anschließend wird versucht, die Daten in das JSON-Format umzuwandeln.

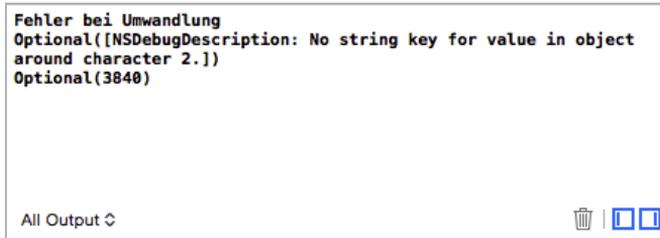
Wenn alles in Ordnung ist, dann wird der *catch*-Abschnitt übersprungen. Wenn nicht, wird der Anwender entsprechend informiert, und es werden noch Fehlermeldungen (Eigenschaft *userInfo*) ausgegeben. Der Code wird natürlich durchlaufen, ohne dass ein Fehler auftritt, da die JSON-Zeichenkette korrekt ist.

```
func errorSample() {
    let jstring = "{ fhsdfhsh }"
    let daten = jstring.data(using: String.Encoding.utf8, allowLossyConversion:
true)
    var nsError = NSError()

    var jresult : Any?
    do {
        jresult = try JSONSerialization.jsonObject(with: daten!, options:
.allowFragments)
    } catch let error as NSError {
        nsError = error
        jresult = nil
    }

    if jresult != nil {
        print("Alles ok!")
    } else {
        print("Fehler bei Umwandlung")
    }
}
```

Um das Beispiel interessant zu gestalten, wird jetzt einmal ein Fehler provoziert. In der Variablen *jstring* wird eine sinnlose, nicht dem JSON-Format entsprechende Zeichenkette abgelegt. Der Umwandlungsversuch mit der Methode *JSONObject* schlägt natürlich fehl. Im Beispiel wird dieses Mal in den *catch*-Abschnitt gesprungen und das *NSError*-Objekt untersucht. Da natürlich ein Problem aufgetreten ist, ist im anschließend untersuchten *NSError*-Objekt auch ein Fehler enthalten. Neben der allgemeinen Fehlermeldung sollen aber noch zusätzliche Informationen ermittelt werden. Hierzu werden die Eigenschaften *userInfo* und *code* des *NSError*-Objekts ausgelesen.



**Abb. 6–12** Auswertung des *NSError*-Objekts im Output-Fenster

Nach dem Start des Programms wird automatisch ein Fehler ausgelöst. Die Ausgabe der Eigenschaften *userInfo* und *code* mit der *print*-Anweisung liefert zusätzliche Informationen im Output-Fenster von Xcode.

### Hinweis

Eine detaillierte Übersicht der Fehlercodes finden Sie unter folgender URL:

[https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation\\_Constants/index.html#//apple\\_ref/doc/constant\\_group/NSError\\_Codes](https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Constants/index.html#//apple_ref/doc/constant_group/NSError_Codes)

Sofern es möglich ist, sollten Sie im Code *NSError*-Objekte immer auswerten.

### Das Protokoll »Error«

Es ist möglich, auch eigene Fehlertypen zu definieren. Im folgenden Beispiel soll das einmal demonstriert werden. Im ersten Schritt wird ein neuer Enumerations-typ angelegt, der vom Protokoll *Error* (wie auch *NSError*) abgeleitet wurde:

```
enum MyBigBadError: Error {
    case small
    case medium
    case big
}
```

Der Enumerationstyp trägt den Namen *MyBigBadError*. Drei Werte (*small*, *medium* und *big*) wurden in ihm definiert. Dieser neue Typ bildet die Basis für die nun folgende Funktion, in der ein Fehler vom Typ *MyBigBadError* auftreten kann:

```
func throwsPossibleError(_ input: Int) throws -> String {
    let result: String

    switch input {
        case 1...3:
            throw MyBigBadError.small
    }
}
```

```

    case 4...6:
        throw MyBigBadError.medium

    case 7...9:
        throw MyBigBadError.big

    default:
        result = "Passed"
}
return result
}

```

Innerhalb der Methode *throwsPossibleError* kann der zuvor definierte Fehler auftreten. Um innerhalb einer Funktion zu kennzeichnen, dass ein Fehler ausgelöst werden kann, wird die Funktion mit dem Schlüsselwort *throws* (nach dem Funktionsnamen) gekennzeichnet. Es handelt sich ja um ein Beispiel. Aus diesem Grund wird eine *switch*-Anweisung verwendet, um einen Fehler vom Typ *MyBigBadError* auszulösen (manche Programmierer sagen wegen »throws« auch: »einen Fehler zu werfen«), wenn nicht ein Integer-Wert größer-gleich 10 der Funktion zuvor als Parameter übergeben wird. Bei übergebenen Werten kleiner 10 wird geprüft, ob diese in eine Kategorie (klein, mittel, groß) passen. Ist das der Fall, wird in den entsprechenden *case*-Zweig gesprungen und mithilfe des Schlüsselwortes *throw* ein Fehler ausgelöst (geworfen). Die eigentliche Aufgabe im *do...try...catch*-Block besteht nun darin, jeden möglichen Fehler abzufangen:

```

func testError() {
    do {
        let result = try throwsPossibleError(5)
        print("Alles ok - es wurde kein Fehler geworfen")
        print(result)
    } catch MyBigBadError.small {
        print("Nur ein kleiner Fehler")
    } catch MyBigBadError.medium {
        print("Ein mittlerer Fehler")
    } catch MyBigBadError.big {
        print("Ein großer Fehler")
    } catch {
        print("Keine Ahnung, was passiert ist!")
    }
}
}

```

Innerhalb der Funktion *testError* wird im *do*-Abschnitt die zuvor definierte Funktion *throwsPossibleError* aufgerufen. Tritt kein Fehler auf, so werden die beiden folgenden *print*-Anweisungen ausgeführt und ausgegeben.

Da als Parameter aber der Wert 5 übergeben wurde, wird innerhalb der Funktion ein Fehler ausgelöst. Im *catch*-Abschnitt wird der Fehler durch die entsprechende Anweisung abgefangen und die zugehörige Fehlermeldung ausgegeben.

Aber auch wenn ein Wert kleiner oder größer 5 der Funktion *throwsPossibleError* übergeben wird, wird eine passende Fehlerbehandlungsroutine aufgerufen und ausgeführt. Für alle möglichen Varianten gibt es einen passenden *catch*-Abschnitt. Selbst für den Fall, dass ein nicht absehbarer Fehler auftritt, ist ein entsprechender Abschnitt zur Fehlerbehandlung (*catch* ohne Parameter) vorhanden. So ausgerüstet, kann dann auch einmal etwas danebengehen.

### Die *defer*-Anweisung

Der Name *defer* leitet sich von »deferred« (engl. für »verzögert« oder »aufgeschoben«) ab. Die Syntax der Anweisung sieht folgendermaßen aus:

```
defer {
    //Anweisung(en)
}
```

Alle Anweisungen in einem *defer*-Block werden zwar verzögert, aber in jedem Fall ausgeführt. Auch wenn nach dem *defer*-Block ein Fehler auftritt, wird der Code im *defer*-Block trotzdem noch abgearbeitet. Dieser Anweisungsblock ist also ideal, um Aktionen auszuführen, die in jedem Fall (z.B. das Schließen einer Datei) vorgenommen werden sollen. Vergleichbares gibt es auch in anderen Sprachen, z.B. in C# oder Java den *finally*-Block. Wenn Sie diese Anweisung bereits kennen, wissen Sie auch, wie *defer* funktioniert. Hier sehen Sie noch einmal das letzte Beispiel, dieses Mal mit einem *defer*-Block:

```
func testError() {
    defer {
        print("defer wurde ausgeführt!")
    }
    do {
        let result = try throwsPossibleError(5)
        print("Alles ok - es wurde kein Fehler geworfen")
        print(result)
    } catch MyBigBadError.small {
        print("Nur ein kleiner Fehler")
    } catch MyBigBadError.medium {
        print("Ein mittlerer Fehler")
    } catch MyBigBadError.big {
        print("Ein großer Fehler")
    } catch {
        print("Keine Ahnung, was passiert ist!")
    }
}
```

Der *defer*-Block steht im Beispiel direkt zu Beginn der Funktion *testError*. Er wird aber nicht umgehend ausgeführt, sondern erst dann, wenn abhängig vom übergebenen Parameter entweder der *do*-Block komplett durchlaufen wurde oder einer

der *catch*-Blöcke ausgeführt worden ist. Wurde die Funktion *throwsPossibleError* mit dem Parameterwert 5 aufgerufen (ein Fehler tritt auf), dann würde die Ausgabe wie folgt aussehen:

```
Ein mittlerer Fehler
defer wurde ausgeführt!
```

Im anderen Fall, die Funktion *throwsPossibleError* wurde mit dem Parameterwert 10 aufgerufen, sieht die Ausgabe so aus:

```
Alles ok – es wurde kein Fehler geworfen
Passed
defer wurde ausgeführt!
```

Ein *defer*-Block kann also sehr gut verwendet werden, um im Abschluss aufzuräumen.

## 6.5 Fehlerbehandlung bei knappem Speicher

In der Regel kümmert sich der ARC-Mechanismus (siehe Abschnitt 4.9) um die Speicherverwaltung unter iOS. Das heißt, sobald der ARC-Zähler bei einem Referenzobjekt feststellt, dass es auf dieses keine Referenzen mehr gibt, wird das Objekt freigegeben. Unter Umständen kann es aber vorkommen, dass der Speicher trotzdem knapp wird. In solchen Fällen informiert iOS die App über einen Aufruf der Methode *didReceiveMemoryWarning*. Diese Methode wird automatisch nach dem Aufruf einer Vorlage jedem Projekt hinzugefügt:

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}
```

Wenn also diese Situation eintritt, dann sollte innerhalb dieser Methode Code vorhanden sein, der im Speicher etwas Freiraum schafft. Wie aber kann ein solcher Code aussehen? Im folgenden Beispiel wird in der Klasse *ViewController* zu Beginn eine Variable *myFoo* vom Typ *Foo* als optional deklariert:

```
import UIKit

class ViewController: UIViewController {

    var myFoo : Foo?

    override func viewDidLoad() {
        super.viewDidLoad()
        myFoo = Foo()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()

        if (myFoo != nil) {
```

```
        myFoo = nil
        print(
            "Der Variablen myFoo wurde nil zugewiesen!")
    } else {
        print("myFoo enthält bereits nil!")
    }
}

print(
    "didReceiveMemoryWarning wurde aufgerufen!")
}
}

class Foo {
}
```

Eine Zuweisung für *myFoo* erfolgt erst zur Laufzeit innerhalb der Methode *viewDidLoad*. Dort wird eine neue Instanz erzeugt und zugewiesen. Meldet iOS jetzt knappen Speicher, so wird die Methode *didReceiveMemoryWarning* aufgerufen. Innerhalb der Methode wird im ersten Schritt geprüft, ob die Variable *myFoo* einen Wert enthält. Sofern das der Fall ist, wird der Variablen *nil* zugewiesen und eine entsprechende Meldung ausgegeben. Enthält die Variable bereits *nil*, dann passiert nichts. In jedem Fall wird via *print* ein Hinweis darüber ausgegeben, dass die Methode aufgerufen wurde.

### Knappen Speicher simulieren

Wie aber simuliert man knappen Speicher? Darauf gibt es zum Glück eine einfache Antwort: innerhalb des iOS-Simulators. Dieser enthält eine entsprechende Funktion.



**Abb. 6-13** Memory Warnings simulieren

Sobald der Simulator gestartet wurde, kann im Menü *Hardware* der entsprechende Menüpunkt aufgerufen werden. Als Folge des Aufrufs wird die Methode *didReceiveMemoryWarning* aufgerufen.

## 6.6 Voraussetzungen prüfen, Fehler vermeiden

Probleme können im Code natürlich auch auftreten, wenn der Quellcode eine bestimmte Version von iOS voraussetzt. Mit Swift 2 ist für diese Situation eine neue Abfrage verfügbar. Mit dem sogenannten *API Availability Checking* lässt sich sehr simpel prüfen, ob die App unter einer bestimmten Version von iOS ausgeführt wird oder nicht. Sie können dafür den folgenden Code nutzen:

```
func isIOS82Available() -> Int? {
    if #available(iOS 8.2, *) {
        return nil
    }
    return 1
}
```

Die Funktion liefert 1 zurück, wenn iOS 8.2 oder eine höhere Version oder auch eine alternative Plattform (z.B. watchOS) verfügbar ist. Die entscheidende Stelle im Code ist *if #available*. Der Code kann natürlich auch in einer *if*-Abfrage verwendet werden, um dann alternativen Code auszuführen:

```
if #available(iOS 9, *) {
    // Anweisungen
} else {
    // Anweisungen
}
```

Das ist aber noch nicht alles, was das *API Availability Checking* ermöglicht. Es können auch Funktionen mit einer entsprechenden Kennzeichnung versehen werden, sodass diese nur unter bestimmten Versionen von iOS aufgerufen werden können:

```
@available(iOS 9, *)
func onlyAvailableUnderiOS9AndGreater() {
    // Anweisung(en)
}
```

Diese Funktion kann nur aufgerufen werden, wenn die App unter iOS 9 ausgeführt wird.

### Überwachte Ausführung mit guard

Manchmal ist es eine gute Idee, Anweisungen nur überwacht ausführen zu lassen. So lässt sich sicherstellen, dass Code direkt, ohne Abbruch, beendet wird, wenn er nicht ausgeführt werden kann. Möglich ist das unter Swift 2 mit dem *guard*-Schlüsselwort. Ähnlich einer *if*-Anweisung prüft *guard* eine Bedingung. Allerdings – und das ist der große Unterschied zur *if*-Anweisung – wird der *guard-else*-Zweig nur ausgeführt, wenn die geprüfte Bedingung nicht zutrifft. Ein (kleines) Beispiel sagt mehr als 1000 Worte. Sehen Sie sich bitte einmal folgenden Code an:

```
func checkNumber(number:Int) {  
    if number <= 5 {  
        return  
    }  
    print("Zahl ist Ok")  
}
```

Innerhalb der Funktion *checkNumber* soll geprüft werden, ob eine übergebene Variable größer als 5 ist. Wenn das der Fall ist, wird eine *print*-Anweisung ausgeführt, wenn nicht, soll der Code bzw. die Funktion beendet werden (mit der *return*-Anweisung). So weit, so gut; die Prüfung mit *<=* ist unschön, aber praktikabel. Nun betrachten wir dieselbe Funktion unter Verwendung der *guard*-Anweisung:

```
func checkNumber(number:Int) {  
    guard number > 5 else {  
        return  
    }  
    print("Zahl ist Ok")  
}
```

Es fällt direkt auf, dass die Prüfung der Bedingung hier genau der Überlegung entspricht. Die ganze Funktion bzw. der darin enthaltene Code wirkt so etwas klarer. Noch deutlicher wird der Vorteil von *guard* aber in Verbindung mit dem API Availability Checking:

```
func testVersion() {  
    guard #available(iOS 9, OSX 10.10, *) else { return }  
    //Anweisungen, die nur unter iOS 8 / OSX 10.10  
    //ausgeführt werden können  
}
```

Der Code innerhalb der Funktion *testVersion* wird eben nur dann ausgeführt, wenn die App ab iOS 9 oder höher oder unter OSX 10.10 und höher ausgeführt wird. Ansonsten wird die Ausführung der Funktion direkt beendet. Klarer geht es nicht. Vor allem dann, wenn entsprechend viele Prüfungen im Code vorhanden sind, wird der Nutzen der Verwendung von *guard* deutlich.

### Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie man Fehler im Programm aufspürt, wie man diese behandelt und wie man ihnen entgegenwirkt.