Erica Sadun

Das große iPhone Entwicklerbuch

Rezepte für Anwendungsprogrammierung mit dem iPhone SDK

ADDISON-WESLEY

3

Objective-C-Trainingslager

Bei der iPhone-Entwicklung arbeiten Sie mit Objective-C, der Standardprogrammiersprache für sowohl das iPhone als auch Mac OS X. Diese Sprache ist sehr leistungsstark und erlaubt Ihnen, beim Schreiben von Anwendungen auf die Frameworks Cocoa und Cocoa Touch von Apple zurückzugreifen. In diesem Kapitel erhalten Sie grundlegende Kenntnisse in Objective-C, um mit der iPhone-Programmierung loslegen zu können. Sie erfahren etwas über Schnittstellen, Methoden, Eigenschaften, Speicherverwaltung usw. Zur Abrundung sehen wir uns neben Objective-C auch noch Cocoa an, um Ihnen die wichtigsten Klassen zu zeigen, die Sie bei Ihrer täglichen Programmierarbeit verwenden werden. Außerdem erhalten Sie konkrete Beispiele zum Einsatz dieser Klassen.

3.1 Die Sprache Objective-C

Objective-C ist eine echte Obermenge von ANSI C. Bei C handelt es sich um eine kompilierte, prozedurale Programmiersprache, die in den frühen 1970er-Jahren von AT&T entwickelt wurde. Objective-C, erdacht von Brad J. Cox, ergänzt C um objektorientierte Merkmale und Prinzipien von Smalltalk-80. Smalltalk ist eine der ältesten und bekanntesten objektorientierten Sprachen und wurde von Xerox PARC entwickelt. Cox hat das Objekt- und Nachrichtensystem auf das standardmäßige C aufgesetzt und so seine neue Sprache geformt. Dadurch können Programmierer weiterhin in der vertrauten Sprache C arbeiten, haben aber in dieser Sprache gleichzeitig Zugriff auf objektorientierte Merkmale. In den späten 1980er-Jahren wurde Objective-C als Hauptentwicklungssprache für das Betriebssystem NeXTStep der neuen Computerfirma NeXT von Steve Jobs übernommen. NeXTStep war die Inspirationsquelle und der Vorläufer von OS X. Die aktuelle Version 2.0 von Objective-C wurde zusammen mit OS X Leopard im Oktober 2007 veröffentlicht.

Bei der objektorientierten Programmierung werden Merkmale eingesetzt, die es im standardmäßigen C nicht gibt. Objekte sind Datenstrukturen mit einer festgelegten Gruppe von Funktionsaufrufen. Jedes Objekt in Objective-C verfügt über *Instanzvariablen*, bei denen es sich um die Felder der Datenstruktur handelt, und über *Methoden*, also die Funktionsaufrufe, die das Objekt ausführen kann. In objektorientiertem Code werden diese Objekte und Methoden verwendet, um in der Programmierung Abstraktionen zu ermöglichen, die die Lesbarkeit und Zuverlässigkeit des Codes verbessern.

Bei der objektorientierten Programmierung können Sie wiederverwendbare Codeeinheiten erstellen, die sich vom normalen Fluss der prozeduralen Entwicklung entkoppeln lassen. Objektorientierte Programme stützen sich nicht auf den Prozessablauf, sondern werden auf der Grundlage intelligenter Datenstrukturen aufgebaut, die aus den Objekten und deren Methoden hervorgehen. Cocoa Touch auf dem iPhone und Cocoa auf Mac OS X bieten eine umfassende Bibliothek solcher intelligenter Objekte. Mit Objective-C haben Sie Zugriff auf diese Bibliothek, sodass Sie den Werkzeugkasten von Apple nutzen können, um mit einem Minimum an Anstrengung und Code leistungsfähige Anwendungen zu schreiben.

Hinweis

Die Klassennamen von Cocoa Touch auf dem iPhone, die mit `NS` beginnen, z. B. `NSString` und `NSArray`, gehen auf NeXT zurück. *NS* steht für NeXTStep, das Betriebssystem der NeXT-Computer.

3.2 Klassen und Objekte

Objekte bilden das Kernstück der objektorientierten Programmierung. Um sie zu definieren, legen Sie Klassen an, die als Vorlagen zur Objekterstellung dienen. In Objective-C gibt eine Klassendefinition an, wie neue Objekte der betreffenden Klasse erstellt werden. Um also ein »Dingsbums«-Objekt anzulegen, definieren Sie die Klasse `Dingsbums` und erstellen damit bei Bedarf neue Objekte.

In jeder Klasse sind die Instanzvariablen und Methoden in einer öffentlichen Headerdatei mit der standardmäßigen C-Endung `.h` aufgeführt. Beispielsweise können Sie für ein Auto ein `Car`-Objekt wie das in *Listing 3.1* erstellen. Die hier gezeigte Headerdatei `Car.h` enthält die Schnittstelle, die deklariert, welche Struktur ein `Car`-Objekt aufweist. Die Namen aller Klassen in Objective-C sollten mit einem großen Anfangsbuchstaben beginnen.

```
#import <Foundation/Foundation.h>
@interface Car : NSObject
{
   int year;
   NSString *make;
   NSString *model;
}
```

```
- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear;
- (void) printCarInfo;
- (int) year;
@end
```

▶ *Listing 3.1: Deklaration der Schnittstelle* `Car` *(*`Car.h`*)*

In Objective-C dient das Symbol `@` zur Kennzeichnung bestimmter Schlüsselwörter. Die beiden entsprechenden Elemente in diesem Listing (`@interface` und `@end`) grenzen Start und Ende der Klassenschnittstellendefinition ab. Die Klassendefinition selbst beschreibt ein Objekt mit den drei Instanzvariablen `year`, `make` und `model`, die wiederum in den geschweiften Klammern zu Beginn der Schnittstelle deklariert werden.

Die Instanzvariable `year` wird als Integer deklariert (mit `int`), `make` und `model` dagegen sind Strings, genauer gesagt, Instanzen von `NSString`. In Objective-C wird eher diese objektorientierte Klasse verwendet als die mit `char *` definierten Bytestrings von C. Wie Sie in diesem Buch immer wieder sehen werden, bietet `NSString` weit mehr Möglichkeiten als C-Strings. Mit dieser Klasse können Sie die Länge eines Strings ermitteln, nach Teilstrings suchen und sie ersetzen, Strings umkehren, Dateierweiterungen herausziehen usw. All diese Möglichkeiten sind in die elementare Objektbibliothek von Cocoa Touch eingebaut.

Diese Klassendefinition deklariert auch drei öffentliche Methoden. Die erste heißt `setMake:andModel:andYear:`, wobei die gesamte dreiteilige Deklaration einschließlich der Doppelpunkte den Namen einer einzigen Methode bildet. In C würden Sie dazu eine Funktion wie `setProperties(char *c1, char *c2, int i)` verwenden. Der Ansatz von Objective-C wirkt zwar etwas sperriger als der von C, bietet aber mehr Klarheit und Selbstdokumentation. Sie müssen nicht raten, was es mit `c1`, `c2` und `i` auf sich hat, da die Verwendung direkt innerhalb des Namens deklariert ist:

```
[myCar setMake:c1 andModel:c2 andYear:i];
```

Die drei Methoden sind als `void`, `void` und `int` typisiert. Wie in C bezieht sich dies auf den Typ der Daten, die die Methode zurückgibt. Die ersten beiden Methoden geben keine Daten zurück, die dritte einen Integer. In C würden die Funktionsdeklarationen der zweiten und dritten Methode `void printCarInfo()` und `int year()` lauten.

Der Ansatz von Objective-C, die Argumente in die Methodennamen einzustreuen, mag für Einsteiger ungewohnt sein, entwickelt sich aber schnell zu einem geschätzten Merkmal. Es ist nicht mehr nötig, zu raten, welches Argument übergeben werden muss, da der Name einer Methode bereits anzeigt, welche Elemente sie entgegennimmt. In Objective-C werden Methodennamen auch als *Selektoren* bezeichnet, und zwar gerade bei der iPhone-Programmierung, vor allem, wenn Sie Aufrufe von `performSelector:` verwenden, um Objekten zur Laufzeit Nachrichten zu senden.

Beachten Sie, dass in der Headerdatei `#import` zum Laden von Headern verwendet wird und nicht `#include`. Beim Importieren von Headern werden in Objective-C automatisch die Dateien übersprungen, die bereits hinzugefügt sind. Dadurch können Sie `#import`-Direktiven in den verschiedenen Quelldateien mehrfach verwenden, ohne Leistungseinbußen zu riskieren.

Hinweis

Den Code für dieses Beispiel und alle anderen Beispiele in diesem Kapitel finden Sie im Beispielcode zu diesem Buch. Informationen darüber, wie Sie diesen Beispielcode aus dem Internet herunterladen können, erhalten Sie im Vorwort.

3.2.1 Objekte erstellen

Um ein Objekt zu erstellen, weisen Sie Objective-C an, Speicher dafür zuzuweisen und einen Zeiger auf das Objekt zurückzugeben. Da Objective-C eine objektorientierte Sprache ist, sieht ihre Syntax etwas anders aus als die des regulären C. Anstatt einfach Funktionen aufzurufen, weisen Sie ein Objekt an, etwas zu tun, und zwar in der Form zweier Elemente in eckigen Klammern. Das erste Element ist das Objekt, das die Nachricht empfängt, das andere die Nachricht selbst: `[objekt nachricht]`.

In unserem Beispiel sendet der Quellcode die Nachricht `alloc` an die Klasse `Car` und dann die Nachricht `init` an das neu zugewiesene `Car`-Objekt. Diese Verschachtelung ist typisch für Objective-C.

```
Car *myCar = [[Car alloc] init];
```

Das Muster »erst zuweisen, dann initiieren«, das Sie hier sehen, ist der übliche Weg zur Instanziierung eines neuen Objekts. Die Klasse `Car` führt die Methode `alloc` aus. Sie weist einen neuen Speicherblock zu, der groß genug ist, um darin alle in der Klassendefinition aufgeführten Instanzvariablen aufzunehmen, setzt die Instanzvariablen auf null und gibt einen Zeiger zurück. Der neu zugewiesene Block wird als *Instanz* bezeichnet und steht für ein einzelnes Objekt im Arbeitsspeicher.

Einige Klassen, z. B. Ansichten, weisen besondere Initialisierer wie `initWithFrame:` auf, Sie können aber auch selbst Initialisierer wie `initWithMake: andModel: andYear:` schreiben. Das Muster, dass zum Erstellen eines neuen Objekts erst eine Zuweisung und dann eine Initialisierung erfolgt, gilt jedoch immer. In jedem Fall erstellen Sie das Objekt im Speicher und setzen dann wichtige Instanzvariablen auf bestimmte Werte.

3.2.2 Speicherzuweisung

In diesem Beispiel wird Speicher von 16 Byte zugewiesen. Wie die Sternchen anzeigen, sind sowohl `make` als auch `model` Zeiger. In Objective-C zeigen Objektvariablen auf das Objekt selbst. Der Zeiger weist eine Größe von vier Byte auf, weshalb `sizeof(myCar)` den Wert 4 zurückgibt. Das Objekt besteht aus zwei Zeigern zu je vier Byte, einem Integer sowie einem zusätzlichen Feld, das nicht aus der Klasse `Car` stammt.

Dieses zusätzliche Feld entspringt der Klasse `NSObject`. In *Listing 3.1* sehen Sie rechts neben dem Doppelpunkt hinter dem Wort `Car` in der Klassendefinition die Angabe `NSObject`. Damit ist `NSObject` die Elternklasse von `Car`, von der `Car` alle Instanzvariablen und Methoden erbt. Das bedeutet, dass `Car` vom Typ `NSObject` ist und alle Speicherzuweisungen erbt, die `NSObject`-Instanzen benötigen. Dies ist der Ursprung der zusätzlichen vier Bytes.

Die Gesamtgröße des zugewiesenen Objekts beträgt 16 Byte. Darin sind die beiden `NSString`-Zeiger mit je vier Byte, ein `int` mit vier Byte und die von `NSObject` geerbte Zuweisung mit vier Byte enthalten. Mit der C-Funktion `sizeof` können Sie die Größe der Objekte auf einfache Weise ausgeben. Im folgenden Code werden mit der Anweisung `printf` des standardmäßigen C Textinformationen an die Konsole gesendet. `printf`-Befehle funktionieren in Objective-C ebenso wie in ANSI C.

```
NSObject *object = [[NSObject alloc] init];
Car *myCar = [[Car alloc] init];

// Gibt 4 zurück, die Größe eines Objektzeigers
printf("object pointer: %d\n", sizeof(object));

// Gibt 4 zurück, die Größe eines NSObject-Objekts
printf("object itself: %d\n", sizeof(*object));

// Gibt 4 zurück, wiederum die Größe eines Objektzeigers
printf("myCar pointer: %d\n", sizeof(myCar));

// Gibt 16 zurück, die Größe eines Car-Objekts
printf("myCar object: %d\n", sizeof(*myCar));
```

3.2.3 Speicher freigeben

In C weisen Sie Speicher mit `malloc()` oder einem ähnlichen Aufruf zu und geben ihn mit `free()` wieder frei; in Objective-C verwenden Sie dazu `alloc` bzw. `release`. (Es gibt in Objective-C auch einige andere Möglichkeiten, um Speicher zuzuweisen, z. B. das Kopieren anderer Objekte.)

```
[object release];
[myCar release];
```

Wie wir in Kapitel 2, *Ein erstes Projekt erstellen*, besprochen haben, ist die Freigabe von Speicher etwas komplizierter als in standardmäßigem C, da Objective-C ein Speichersystem mit Referenzzähler verwendet. Jedes Objekt im Speicher weist einen Beibehaltungszähler auf, den Sie einsehen können, indem Sie dem Objekt die Nachricht `retainCount` senden. Bei seiner Erstellung hat jedes Objekt den Wert 1 in seinem Beibehaltungszähler. Dieser Zähler wird jedes Mal um 1 verringert, wenn das Objekt die Nachricht `release` empfängt. Erreicht der Zähler für ein Objekt den Wert 0, wird es in den allgemeinen Speicherpool freigegeben.

```
// Der Beibehaltungszähler beträgt nach dem Erstellen 1
printf("The retain count is %d\n", [myCar retainCount]);

// Hierdurch wird der Beibehaltungszähler auf 0 herabgesetzt
[myCar release];
```

```
// Der folgende Befehl führt zu einem Fehler, da das Objekt bereits
// freigegeben ist
printf("Retain count is now %d\n", [myCar retainCount]);
```

Wenn Sie Nachrichten an freigegebene Objekte senden, bringen Sie Ihre Anwendung zum Absturz. Bei der Ausführung des zweiten `printf`-Befehls geht die `printf`-Nachricht an das bereits freigegebene `myCar`-Objekt. Dies führt zu einer Speicherzugriffsverletzung, weshalb das Programm abgebrochen wird.

```
The retain count is 1
objc[10754]: FREED(id): message retainCount sent to freed
object=0xd1e520
```

Auf dem iPhone gibt es keine Garbage Collection, weshalb Sie als Entwickler Ihre Objekte selbst verwalten müssen. Erhalten Sie sie so lange aufrecht, wie sie verwendet werden, und geben Sie den Speicher frei, wenn Sie sie nicht mehr benötigen. Weitere Informationen über grundlegende Verfahren zur Speicherverwaltung finden Sie weiter hinten in diesem Kapitel.

3.3 Methoden, Nachrichten und Selektoren

Zum Zuweisen und Initialisieren von Daten verwenden Sie in Standard-C zwei Funktionsaufrufe. Im Gegensatz zur Anweisung `[[Car alloc] init]` von Objective-C sieht das wie folgt aus:

```
Car *myCar = malloc(sizeof(Car));
init(myCar);
```

In Objective-C wird die Syntax *`funktionsname(argument)`* nicht verwendet. Stattdessen senden Sie in der Schreibweise mit eckigen Klammern Nachrichten an Objekte. Eine solche Nachricht weist das Objekt an, eine Methode auszuführen. Die Verantwortung dafür, die Methode zu implementieren und ein Ergebnis hervorzurufen, liegt beim Objekt. Das erste Element innerhalb der eckigen Klammern ist der Empfänger der Nachricht, das zweite ein Methodenname, unter Umständen mit Argumenten für diese Methode. Zusammengenommen bildet dies die Nachricht, die Sie senden möchten.

In C schreiben Sie beispielsweise Folgendes:

```
printCarInfo(myCar);
```

In Objective-C dagegen heißt das:

```
[myCar printCarInfo];
```

Trotz der Unterschiede in der Syntax sind Methoden im Grunde genommen Funktionen, die auf Objekten operieren. Sie weisen auch die gleichen Typen auf, die in Standard-C zur Verfügung stehen. Anders als bei Funktionsaufrufen gibt es in Objective-C jedoch Einschränkungen dafür, wer Methoden implementieren und aufrufen darf. Methoden gehören zu Klassen, und die Klassenschnittstelle definiert, welche dieser Methoden für die Außenwelt deklariert ist.

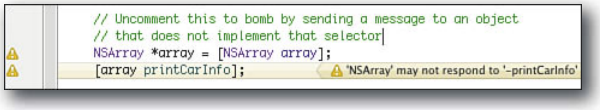
3.3.1 Dynamische Typisierung

Neben der statischen Typisierung wird in Objective-C auch eine dynamische verwendet. Bei der statischen Typisierung wird eine Variablendeklaration zur Kompilierungszeit auf eine bestimmte Klasse beschränkt. Dagegen ist bei der dynamischen Typisierung das Laufzeitsystem und nicht der Compiler dafür verantwortlich, die Objekte zu fragen, welche Methoden sie ausführen können und zu welcher Klasse sie gehören. Das bedeutet, dass während der Programmausführung ausgewählt werden kann, welche Nachrichten gesendet werden und an welche Objekte sie gehen. Dieses Merkmal bietet sehr viele Möglichkeiten und wird normalerweise mit Interpreter-Systemen wie Lisp verbunden. Sie können ein Objekt auswählen, im Programm eine Nachricht anlegen und diese Nachricht an das Objekt senden, ohne zu wissen, welches Objekt zur Kompilierungszeit ausgewählt wird und welche Nachricht ihm gesendet wird.

Mit diesen Möglichkeiten geht natürlich auch Verantwortung einher. Sie können Nachrichten nur an Objekte senden, die die im Selektor beschriebene Methode auch tatsächlich implementieren (es sei denn, die Klasse kann Nachrichten verarbeiten, die keine Implementierungen aufweisen – dazu müssen Sie die Aufrufweiterleitung von Objective-C verwenden, die weiter hinten in diesem Kapitel besprochen wird). Wenn Sie `printCarInfo` beispielsweise an ein Arrayobjekt senden, führt dies zu einem Laufzeitfehler und dem Absturz des Programms, denn in Arrays ist diese Methode nicht definiert. Nur Objekte, die die gegebene Methode implementieren, können korrekt auf die Nachricht reagieren und den angeforderten Code ausführen.

```
2009-05-08 09:04:31.978 HelloWorld[419:20b] *** -[NSCFArray printCarInfo]:
➥unrecognized selector sent to instance 0xd14e80
2009-05-08 09:04:31.980 HelloWorld[419:20b] *** Terminating app due to
➥uncaught exception 'NSInvalidArgumentException', reason: ,*** -[NSCFArray
➥printCarInfo]: unrecognized selector sent to instance 0xd14e80'
```

Bei der Kompilierung überprüft Objective-C anhand der statischen Typisierung der Objekte die an sie gesendeten Nachrichten. Die Arraydefinition in *Listing 3.1* ist statisch deklariert und teilt dem Compiler mit, dass das fragliche Objekt vom Typ `(NSArray *)` ist. Wenn der Compiler Objekte findet, die auf die angeforderten Methoden möglicherweise nicht reagieren können, gibt er eine Warnung aus.



▶ *Abbildung 3.1: Der Objective-C-Compiler von Xcode gibt eine Warnung aus, wenn er eine Methode findet, die der Empfänger anscheinend nicht implementiert.*

Diese Warnungen bedeuten nicht, dass die Kompilierung fehlschlägt, Es wäre auch durchaus möglich, dass `NSArray` die Methode `printCarInfo` tatsächlich implementiert, diese Implementierung aber lediglich nicht in der veröffentlichten Schnittstelle deklariert hätte, sodass der Code trotzdem problemlos läuft. Da `NSArray` die Methode jedoch wirklich nicht implementiert hat, führt die Ausführung dieses Codes zur Laufzeit zu dem bereits gezeigten Absturz.

Aufgrund der dynamischen Typisierung können Sie in Objective-C auf verschiedene Art und Weise auf ein und dieselbe Art von Objekt zeigen. Zwar wurde `array` als statisch typisiertes `(NSArray *)`-Objekt deklariert, doch nutzt es die gleichen internen Objektdatenstrukturen wie ein als `id` deklariertes Objekt. Der Typ `id` kann auf jedes Objekt unabhängig von seiner Klasse zeigen und ist mit `(NSObject *)` gleichwertig. Die folgende Zuweisung ist gültig und ruft keinerlei Warnungen zur Kompilierungszeit hervor:

```
NSArray *array = [NSArray array];
// Diese Zuweisung ist gültig
id untypedVariable = array;
```

Zur weiteren Veranschaulichung betrachten Sie ein veränderbares Array. `NSMutableArray` ist eine Unterklasse von `NSArray` und ermöglicht es, Arrays zu erstellen, die sich ändern und bearbeiten lassen. Wenn Sie ein veränderbares Array erstellen und typisieren, es aber zu einem Arrayzeiger zuweisen, wird dies fehlerfrei kompiliert. Im folgenden Beispiel ist `anotherArray` zwar statisch als `NSArray` typisiert, doch wenn Sie es wie folgt erstellen, wird dadurch zur Laufzeit ein Objekt erstellt, das alle Instanzvariablen und Verhalten der Klasse für veränderbare Arrays enthält.

```
NSArray *anotherArray = [NSMutableArray array];
// Dieser Aufruf einer Methode, die nur für veränderbare Arrays gilt, ist
// gültig, führt bei der Kompilierung aber zu einer Warnung
[anotherArray addObject:@"Hello World"];
```

Was hier zu einer Warnung führt, ist nicht die Erstellung und Zuweisung, sondern die Verwendung. Wenn wir `addObject:` an `anotherArray` senden, nutzen wir unser Wissen darüber aus, dass dieses Array in Wirklichkeit veränderbar ist, auch wenn es statisch als `(NSArray *)` typisiert wurde. Dies kann der Compiler jedoch nicht erkennen, weshalb die Verwendung zur Kompilierungszeit zu der Warnung »`'NSArray'` may not respond to `-addObject:`'« (»`'NSArray'` kann möglicherweise nicht auf `-addObject:` reagieren«) führt. Zur Laufzeit jedoch funktioniert der Code fehlerlos.

Das Objekt einer Kindklasse zu einem Zeiger der Elternklasse zuzuordnen, funktioniert zur Laufzeit zwar im Allgemeinen, doch den umgekehrten Weg zu gehen ist weit gefährlicher. Jedes veränderbare Array ist eine Art von Array und kann daher alle Nachrichten empfangen, die für Arrays gelten. Dagegen ist nicht jedes Array veränderbar. Wenn Sie die Nachricht `addObject:` an ein reguläres Array senden, ist das fatal. Zur Laufzeit ist die Katastrophe da, weil Arrays diese Methode nicht implementieren.

```
NSArray *standardArray = [NSArray array];
NSMutableArray *mutableArray;
// Diese Zeile führt zu einer Warnung
mutableArray = standardArray;
// Diese Zeile führt zur Laufzeit zu einem Desaster
[mutableArray addObject:@"Hello World"];
```

Der hier gezeigte Code ruft nur eine Warnung hervor, nämlich in der Zeile, in der das standardmäßige Arrayobjekt einem Zeiger für veränderbare Arrays zugeordnet wird. Die Warnung lautet: »assignment from distinct Objective-C type« (»Zuweisung von gesondertem Objective-C-Typ«). Zuweisungen von

Eltern- zu Kindobjekten führen nicht zu dieser Warnung, aber Zuweisungen von Kind- zu Elternobjekten. Nehmen Sie Zuweisungen also nur zwischen Klassen vor, die völlig unabhängig voneinander sind. Ignorieren Sie diese Warnungen nicht, sondern korrigieren Sie den Code, da Sie anderenfalls das Fundament für einen Absturz zur Laufzeit legen. Objective-C ist eine kompilierte Sprache mit dynamischer Typisierung und führt viele der Laufzeitüberprüfungen nicht durch, die interpretierte objektorientierte Sprachen vornehmen.

Hinweis

Den Compiler von Xcode können Sie so einrichten, dass er Warnungen als Fehler behandelt, indem Sie das Flag `GCC_TREAT_WARNINGS_AS_ERRORS` im Fenster **Project Info ▸ Build ▸ User-Defined** setzen. Da Objective-C so dynamisch ist, kann der Compiler im Gegensatz zu Compilern statischer Sprachen nicht jedes Problem erfassen, das zur Laufzeit zu einem Absturz führen kann. Achten Sie also auf die Warnungen, und versuchen Sie, deren Ursachen zu beseitigen.

3.3.2 Methoden erben

Objekte erben sowohl Methodenimplementierungen als auch Instanzvariablen. Ein `Car`-Objekt ist eine Art von `NSObject`-Objekt und kann daher auf die gleichen Nachrichten reagieren wie Letzteres. Aus diesem Grunde kann `myCar` mit `alloc` und `init` zugewiesen und initialisiert werden: Diese Methoden werden durch `NSObject` definiert und können daher von allen Instanzen von `Car` verwendet werden, da `Car` von der Klasse `NSObject` abgeleitet ist.

Ebenso sind alle `NSMutableArray`-Instanzen eine Art von `NSArray`-Objekt. Alle Methoden von Arrays können auch von veränderbaren Arrays und deren Kindklassen verwendet werden. Sie können die Elemente in einem Array zählen, ein Objekt anhand seiner Indexzahl abrufen usw.

Eine Kindklasse kann die Implementierung einer Methode ihrer Elternklasse überschreiben, aber das Vorhandensein der Methode nicht aufheben. Kindklassen erben stets sämtliche Verhalten und Eigenschaften ihrer Elternklassen.

3.3.3 Methoden deklarieren

Wie *Listing 3.1* zeigt, definiert eine Klassenschnittstelle die Instanzvariablen und Methoden, die eine neue Klasse neben denen ihrer Elternklasse hat. Die Schnittstelle befindet sich normalerweise in einer Headerdatei mit der Erweiterung `.h`. In *Listing 3.1* hat die Schnittstelle die drei folgenden Methoden deklariert:

```
- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear;
- (void) printCarInfo;
- (int) year;
```

Die drei Methoden geben `void`, `void` und `int` zurück. Beachten Sie den Bindestrich zu Beginn der Methodendeklaration. Er zeigt an, dass die Methoden in den Objektinstanzen implementiert werden. So rufen Sie beispielsweise `[myCar year]` auf und nicht `[Car year]`, denn anderenfalls würde die Nachricht an die Klasse `Car` gesendet werden und nicht an ein `Car`-Objekt. Erläuterungen zu Klassenmethoden (die durch + statt durch - gekennzeichnet sind) folgen weiter hinten in diesem Abschnitt.

Wie ich bereits erwähnt habe, können Methodenaufrufe kompliziert werden. Der folgende Aufruf sendet eine Methodenanforderung mit drei Parametern, die in den Methodenaufruf eingeflochten sind. Der Name der Methode, also ihr Selektor, lautet `setMake: andModel: andYear:`, wobei die drei Doppelpunkte anzeigen, wo die Parameter eingefügt werden müssen. Die Typen der einzelnen Parameter sind in der Schnittstelle hinter den Doppelpunkten definiert, und zwar als `(NSString *)`, `(NSString *)` und `(int)`. Da diese Methode `void` zurückgibt, werden die Ergebnisse keiner Variable zugewiesen.

```
[myCar setMake:@"Ford" andModel:@"Prefect" andYear:1946];
```

3.3.4 Methoden implementieren

Das Paar aus Methoden- und Headerdatei zusammen enthält alle notwendigen Informationen, um eine Klasse zu implementieren und für den Rest der Anwendung bekannt zu machen. Der Implementierungsabschnitt einer Klassendefinition umfasst den Code zur Implementierung der Funktionalität. Dieser Quelltext befindet sich gewöhnlich in einer `.m`-Datei (für »Methode«).

Listing 3.2 zeigt die Implementierung für das Beispiel der Klasse `Car`. Es umfasst den Code für alle drei in der Headerdatei von *Listing 3.1* deklarierten Methoden und fügt eine vierte hinzu. Diese zusätzliche Methode definiert `init` neu. Die `Car`-Version von `init` setzt `make` und `model` auf `nil`, den NULL-Zeiger für Objective-C-Objekte. Außerdem initialisiert diese Methode das Baujahr auf 1901.

Die Sondervariable `self` verweist auf das Objekt, das die Methode implementiert, und wird durch das zugrunde liegende Laufzeitsystem von Objective-C bereitgestellt. In diesem Fall verweist `self` auf die aktuelle Instanz der Klasse `Car`. Der Aufruf `[self nachricht]` weist Objective-C an, eine Nachricht an das Objekt zu senden, das zurzeit die Methode ausführt.

Zu der hier gezeigten Methode `init` gibt es einiges zu bemerken. Erstens gibt sie einen Wert vom Typ `(id)` zurück. Wie ich in diesem Kapitel bereits erwähnt habe, ist der Typ `id` mehr oder weniger gleichwertig mit `(NSObject *)`, allerdings zumindest theoretisch etwas allgemeiner gefasst. Dieser Typ kann auf jegliches Objekt jeglicher Klasse zeigen (auch auf `Class`-Objekte selbst). Resultate geben Sie wie in C mit `return` zurück. Der Zweck von `init` besteht darin, eine korrekt initialisierte Version des Empfängers mithilfe von `return self` zurückzugeben.

Zweitens ruft die Methode `[super init]` auf. Dadurch wird Objective-C angewiesen, eine Nachricht an eine andere Implementierung zu senden, nämlich an die in der Oberklasse des Objekts. Die Oberklasse von `Car` ist `NSObject`, wie aus *Listing 3.1* hervorgeht. Dieser Aufruf besagt: »Bitte führe die Initialisierung durch, die meine Elternklasse normalerweise erledigt, und wende erst dann mein eigenes Verhalten an.«

Schließlich ist noch die Überprüfung `if (!self)` bemerkenswert. In seltenen Fällen können Speicherprobleme auftreten, sodass der Aufruf `[super init]` den Wert `nil` zurückgibt. Dann aber gibt die Methode `init` die Steuerung zurück, ohne irgendwelche Instanzvariablen zu setzen. Da ein `nil`-Objekt nicht auf zugewiesenen Speicher zeigt, ist es nicht möglich, über `nil` auf Instanzvariablen zuzugreifen.

Die anderen Methoden verwenden `year`, `make` und `model`, als ob dies lokal deklarierte Variablen wären. Als Instanzvariablen sind sie innerhalb des Kontextes des aktuellen Objekts definiert und können wie in diesem Beispiel gezeigt gesetzt und gelesen werden. Die Methode `UTF8String`, die an die Instanzvariablen `make` und `model` gesendet wird, konvertiert diese `NSString`-Objekte in C-Strings, die mit dem Formatspezifizierer `%s` ausgegeben werden können.

Hinweis

Sie können beliebige Nachrichten an `nil` senden, z. B. `[nil beliebigeMethode]`. Das Ergebnis ist wiederum `nil` (oder genauer gesagt, 0 umgewandelt in `nil`). Mit anderen Worten: Dieser Vorgang hat keinerlei Auswirkungen. Dank dieses Verhaltens können Sie Methodenaufrufen mit einer Absicherung für den Fall verschachteln, dass eine der einzelnen Methoden fehlschlägt und `nil` zurückgibt. Wenn während der Zuweisung mit `[[Car alloc] init]` der Speicher ausgeht, wird die `init`-Nachricht an `nil` gesendet, wodurch die gesamte `alloc`/`init`-Anforderung `nil` zurückgibt.

```
#import "Car.h"

@implementation Car
- (id) init
{
   self = [super init];
   if (!self) return nil;

   make = nil;
   model = nil;
   year = 1901;

   return self;
}

- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear
{
   make = [NSString stringWithString:aMake];
   model = [NSString stringWithString:aModel];
   year = aYear;
}
```

```
- (void) printCarInfo
{
   if (!make) return;
   if (!model) return;

   printf("Car Info\n");
   printf("Make: %s\n", [make UTF8String]);
   printf("Model: %s\n", [model UTF8String]);
   printf("Year: %d\n", year);
}

- (int) year
{
   return year;
}
@end
```

▶ *Listing 3.2: Implementierung der Klasse* `Car` *(*`Car.m`*)*

3.3.5 Klassenmethoden

Klassenmethoden werden mit dem Pluszeichen (+) als Präfix definiert statt mit dem Bindestrich (-). Deklaration und Implementierung erfolgen aber wie bei Instanzmethoden. Beispielsweise können Sie einer Schnittstelle die folgende Methodendeklaration hinzufügen:

```
+ (NSString *) motto;
```

In der Implementierung kodieren Sie sie dann wie folgt:

```
+ (NSString *) motto
{
   return(@"Ford Prefects are Mostly Harmless");
}
```

Klassenmethoden unterscheiden sich von Instanzmethoden darin, dass sie generell keinen Status verwenden können. Das heißt, sie haben keinen Zugriff auf Instanzvariablen, da diese Elemente nur beim Zuweisen von Objekten im Speicher erstellt werden.

Warum also sollte man überhaupt Klassenmethoden verwenden? Dazu gibt es drei Gründe. Erstens liefern Klassenmethoden Ergebnisse, ohne dass ein Objekt instanziiert werden muss. Die Methode `motto` ruft ein hartkodiertes Ergebnis hervor, das nicht vom Zugriff auf Variablen abhängt. Hilfsmethoden wie diese sind in Klassen häufig besser aufgehoben als in der Form von Instanzmethoden.

Stellen Sie sich eine Klasse für geometrische Operationen vor. In dieser Klasse können Sie eine Umrechnung zwischen Bogenmaß und Grad implementieren, ohne dass dafür eine Instanz erforderlich ist, z. B. in der Form `[GeometryClas convertAngleToRadians:theta];`. C-Funktionen, die in der Headerdatei deklariert sind, eignen sich auch sehr gut für diese Aufgabe.

Der zweite Grund besteht darin, dass Klassenmethoden Singletons verbergen können. Dabei handelt es sich um statisch zugewiesene Instanzen, die im iPhone SDK massenhaft vorhanden sind. So gibt `[UIApplication sharedApplication]` z. B. einen Zeiger auf das Singleton-Objekt zurück, das Ihre Anwendung ist. `[UIDevice currentDevice]` ruft ein Objekt ab, das für die Hardwareplattform steht, auf der Sie arbeiten.

Durch die Kombination einer Klassenmethode mit einem Singleton können Sie von überall in Ihrer Anwendung auf diese statische Instanz zugreifen, ohne dazu einen Zeiger auf das Objekt oder eine Instanzvariable zu dessen Speicherung zu benötigen. Die Klassenmethode ruft die Referenz des Objekts ab und gibt sie auf Verlangen zurück.

Drittens fügen sich Klassenmethoden in Speicherverwaltungsverfahren ein. Nehmen wir an, Sie wollen ein neues `NSArray` zuweisen. Dies können Sie mit `[[NSArray alloc] init]` erledigen, aber auch mit `[NSArray array]`. Die Klassenmethode im zweiten Fall gibt ein Arrayobjekt zurück, das initialisiert und für die automatische Freigabe (*autorelease*) eingerichtet wurde. Wie Sie in diesem Kapitel noch lesen werden, hat Apple einen Standard für Klassenmethoden aufgestellt, die Objekte erstellen, sodass diese Methoden die Objekte stets mit automatischer Freigabe zurückgeben. Daher ist die Verwendung von Klassenmethoden ein elementarer Bestandteil des standardmäßigen Speicherverwaltungssystems auf dem iPhone.

3.3.6 Schnelle Auflistung

Die schnelle Auflistung wurde in Objective-C 2.0 eingeführt und bietet eine einfache und elegante Möglichkeit, um Sammlungen wie Arrays und Mengen aufzulisten. Hierbei wird zusätzlich eine `for`-Schleife verwendet, die die Sammlung mit einer knappen `for/in`-Syntax durchläuft. Die Auflistung ist sehr effizient und erfolgt rasch, außerdem ist sie sicher. Versuche, die Sammlung zu ändern, während sie aufgelistet wird, führen zu einer Laufzeitausnahme.

```
NSArray *colors = [NSArray arrayWithObjects:
   @"Black", @"Silver", @"Gray", nil];
for (NSString *color in colors)
   printf("Consider buying a %s car \r\n", [color UTF8String]);
```

Hinweis

Seien Sie vorsichtig, wenn Sie Methoden wie `arrayWithObjects:` oder `dictionaryWithKeysAndValues:` verwenden, da sie recht fehleranfällig sind. Häufig verwenden Entwickler diese Methoden für Instanzvariablen, ohne zuerst sicherstellen, dass die Werte nicht `nil` sind.

3.4 Klassenhierarchie

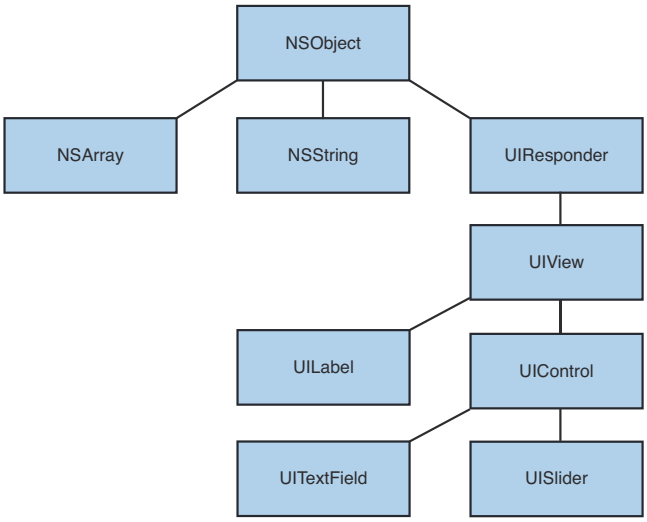
In Objective-C wird jede neue Klasse von einer bereits bestehenden abgeleitet. Die in den *Listings 3.1* und *3.2* beschriebene Klasse `Car` ist auf `NSObject` aufgebaut, der Wurzelklasse des Objective-C-Klassenbaums. Jede Unterklasse erbt Eigenschaften und Verhalten von ihrer Eltern- oder Oberklasse

und verändert sie bzw. fügt eigene Eigenschaften und Verhalten hinzu. Beispielsweise ergänzt die Klasse `Car` das geerbte gewöhnliche `NSObject` um mehrere Instanzvariablen und Methoden.

Abbildung 3.2 zeigt einige der Klassen auf dem iPhone sowie die Beziehungen zwischen ihnen in der Klassenhierarchie. Strings und Arrays stammen ebenso von `NSObject` ab wie die Klasse `UIResponder`, wobei `UIResponder` der Urahn aller iPhone-Bildschirmelemente ist. Ansichten, Beschriftungen, Textfelder und Schieberegler sind Kinder, Enkel und weitere Nachkommen von `UIResponder` und `NSObject`.

Jede Klasse außer `NSObject` stammt von einer anderen Klasse ab. `UITextField` ist eine Art von `UIControl`, das wiederum eine Art von `UIView` ist, und dies wiederum ist ein `UIResponder`, der selbst ein `NSObject` ist. In Objective-C dreht sich alles darum, neue Dinge innerhalb dieser Objekthierarchie einzufügen. Kindklassen können Folgendes tun:

- Sie können neue Instanzvariablen hinzufügen, die ihre Eltern- oder Oberklasse nicht zuweist. Die Klasse `Car` ergänzt drei davon, nämlich die Strings `make` und `model` sowie den Integer `year`.
- Sie können neue Methoden hinzufügen, die in der Elternklasse nicht definiert sind. In `Car` sind mehrere neue Methoden definiert, mit denen Sie die Werte der Instanzvariablen setzen und einen Bericht über das Auto ausgeben können.
- Sie können Methoden überschreiben, die von der Elternklasse definiert werden. Die Methode `init` der Klasse `Car` überschreibt die Version von `NSObject`. Wenn einem `Car`-Objekt eine `init`-Nachricht gesendet wird, führt es seine eigene Version der Methode aus, nicht die von `NSObject`. Gleichzeitig sorgt der Code für `init` dafür, dass die Methode `init` von `NSObject` via `[super init]` aufgerufen wird. Die Elternimplementierung zu erweitern und gleichzeitig auf sie zu verweisen ist eines der grundlegenden Entwurfsprinzipien von Objective-C.



▶ *Abbildung 3.2: Alle Cocoa Touch-Klassen stammen von* `NSObject` *ab, der Wurzel des Klassenhierarchiebaums.*

3.5 Informationen festhalten

Nachdem Sie jetzt die Grundlagen zu Klassen und Objekten kennen, müssen Sie wissen, wie Sie Informationen darüber festhalten. Neben `printf` bietet Objective-C die grundlegende Protokollierungsfunktion `NSLog`. Sie funktioniert ähnlich wie `printf`, die Ausgabe erfolgt aber in `stderr` statt in `stdout`. Außerdem verwendet `NSLog` einen `NSString`-Formatstring statt eines C-Strings.

`NSString`s werden anders deklariert als C-Strings, nämlich mit einem voranstehenden @-Symbol. Ein typischer `NSString` sieht `@"wie folgt"` aus, der gleichwertige C-String dagegen `"wie folgt"`, also ohne das @. Während C-Strings auf einen Zeiger zu einem Bytestring verweisen, handelt es sich bei `NSString`s um Objekte. Einen C-String können Sie bearbeiten, indem Sie die in den einzelnen Bytes gespeicherten Werte ändern, `NSString`s dagegen sind nicht veränderbar. Sie haben keinen Zugriff auf die Bytes, um sie zu bearbeiten, und die eigentlichen Stringdaten sind nicht innerhalb des Objekts gespeichert.

```
// Dies sind 12 Bytes adressierbaren Speichers
printf("%d\n", sizeof("Hello World"));

// Dies ist ein 4-Byte-Objekt, das auf nicht adressierbaren Speicher zeigt
NSString *string = @"Hello World";
printf("%d\n", sizeof(*string));
```

`NSLog` verwendet nicht nur die standardmäßigen C-Formatspezifizierer, sondern führt auch den Objektspezifizierer `%@` ein, mit dem sich Objekte ausgeben lassen. Damit können Sie

```
printf("Make: %s\n", [make UTF8String]);
```

in folgende Form umschreiben:

```
NSLog(@"Make: %@", make);
```

Tabelle 3.1 zeigt einige der häufigsten Formatspezifizierer. Diese Liste ist jedoch alles andere als vollständig. Weitere Einzelheiten erfahren Sie im *String Programming Guide for Cocoa* von Apple (*http://developer.apple.com/mac/library/documentation/cocoa/conceptual/Strings*).

Spezifizierer	Bedeutung
`%@`	Objective-C-Objekt, das die Ergebnisse von `description` oder `descriptionWithLocale:` verwendet
`%%`	Das Literalzeichen „%“
`%d`	Integer mit Vorzeichen (32 Bit)
`%u`	Integer ohne Vorzeichen (32 Bit)
`%f`	Fließkommazahl (64 Bit)
`%e`	Fließkommazahl in wissenschaftlicher Notation mit Exponent (64 Bit)

%c	Zeichendaten ohne Vorzeichen (8 Bit)
%C	Unicode-Zeichendaten (16 Bit)
%s	Zeichendatenarray mit NULL-Terminierung (String, 8 Bit)
%S	Unicode-Zeichendatenarray mit NULL-Terminierung (16 Bit)
%p	Zeigeradresse mit Hexausgabe in Kleinbuchstaben und führendem 0x
%x	Hexwert mit Kleinbuchstaben ohne Vorzeichen (32 Bit)
%X	Hexwert mit Großbuchstaben ohne Vorzeichen (32 Bit)

▶ *Tabelle 3.1: Häufige Stringformatspezifizierer*

NSLog braucht kein hartkodiertes Wagenrücklaufzeichen, sondern hängt automatisch eine neue Zeile an. Darüber hinaus fügt es auch jedem Protokoll einen Zeitstempel hinzu, sodass das Ergebnis des zuvor gezeigten NSLog-Ausrufs wie folgt aussieht:

```
2009-05-07 14:19:08.792 HelloWorld[11197:20b] Make: Ford
```

Über die Nachricht description wird fast jedes Objekt in einen String verwandelt. NSLog verwendet description, um den Inhalt der mit %@ formatierten Objekte zu zeigen. Dabei wird ein NSString mit einer Beschreibung des Empfängerobjekts in Textform zurückgegeben. Sie können Objekte auch außerhalb von NSLog beschreiben, indem Sie ihnen die Nachricht description senden. Dies ist vor allem im Gegensatz zu den Befehlen printf und fprintf nützlich, die sonst keine Objekte ausgeben können.

```
fprintf(stderr, "%s\n", [[myCar description] UTF8String]);
```

Eine weitere hilfreiche Protokollierungsfunktion ist CFShow(). Sie nimmt ein Argument entgegen, nämlich ein Objekt, und gibt eine Beschreibung des Objekts als Momentaufnahme an stderr aus.

```
CFShow(make);
```

Wie NSLog sendet auch CFShow eine description-Nachricht an die Objekte, die es anzeigt, überflutet im Gegensatz zu NSLog aber nicht Ihre Debuggerkonsole mit Zeitstempeln. Damit eignet sich diese Funktion für diejenigen, die lieber auf diese zusätzlichen Informationen verzichten. Für CFShow sind keine Formatierungsstrings erforderlich, was es einfacher macht, diese Funktion zum Code hinzuzufügen. Sie kann aber nur für Objekte eingesetzt werden, nicht für Integer- oder Fließkommawerte.

3.6 Eigenschaften

Eigenschaften legen Variablen und Methoden über sogenannte *Zugriffsmethoden* nach außen offen, also über Methoden, die auf Informationen zugreifen. Die Verwendung von Eigenschaften scheint überflüssig zu sein, da in der Klassendefinition von *Listing 3.1* bereits öffentliche Methoden bekannt gemacht werden. Wozu braucht man also noch Eigenschaften? Die Verwendung von Eigenschaften hat jedoch einige Vorteile gegenüber dem Einsatz selbst gestrickter Methoden, vor allem die Punktschreibweise und die Speicherverwaltung.

3.6.1 Punktschreibweise

Dank der Punktschreibweise können Sie auf Objektinformationen zugreifen, ohne Klammern verwenden zu müssen. Anstatt `[myCar year]` aufzurufen, um die Instanzvariable `year` abzurufen, schreiben Sie `myCar.year`. Das mag zwar so aussehen, als würden Sie unmittelbar auf die Instanzvariable zugreifen, doch das ist nicht der Fall. Eigenschaften rufen stets Methoden auf, die wiederum auf die Daten eines Objekts zugreifen. Tatsächlich verletzen Sie damit die Kapselung von Objekten nicht, da die Eigenschaften auf Methoden zurückgreifen, um die Daten aus dem Objekt herauszuholen.

Da diese Methoden verborgen sind, wird das Erscheinungsbild Ihres Codes bei der Verwendung von Eigenschaften einfacher. Um z. B. den Text einer Tabellenzelle mithilfe von Eigenschaften zu setzen, gehen Sie wie folgt vor:

```
myTableViewCell.textLabel.text = @"Hello World";
```

Das ist weniger mühselig als die folgende Variante ohne Eigenschaften:

```
[[myTableViewCell textLabel] setText:@"Hello World"];
```

Die Codeversion mit Eigenschaften lässt sich leichter lesen und deutlich einfacher warten.

3.6.2 Eigenschaften und Speicherverwaltung

Eigenschaften vereinfachen die Speicherverwaltung. Sie können Eigenschaften erstellen, die Instanzvariablen während der Lebensdauer der Objekte automatisch beibehalten und sie freigeben, sobald Sie diese Variablen auf `nil` setzen. Wenn Sie eine beibehaltene Eigenschaft setzen, wird der Speicher erst dann freigegeben, wenn Sie dies anfordern.

Die Methode `arrayWithObjects:` gibt normalerweise ein Objekt mit automatischer Freigabe zurück, dessen Speicherzuweisung am Ende des Zyklus der Ereignisschleife aufgehoben wird. (Einzelheiten über Autorelease-Pools finden Sie in Kapitel 1, *Einführung in das iPhone SDK*. Eine ausführlichere Erläuterung der Speicherverwaltung folgt in diesem Kapitel.) Wenn Sie das Array zu einer beibehaltenen Eigenschaft zuweisen, bleibt es dadurch dauerhaft bestehen:

```
self.colors = [NSArray arrayWithObjects:
   @"Gray", @"Silver", @"Black"];
```

Wenn Sie das Array nicht mehr brauchen und seinen Speicher freigeben möchten, setzen Sie die Eigenschaft auf `nil`. Dies funktioniert, da Objective-C Zugriffsmethoden synthetisieren kann, um die Änderung der Werte von Instanzvariablen korrekt zu verwalten. In Wirklichkeit setzen Sie die Variable gar nicht auf `nil`, sondern weisen Objective-C an, eine Methode auszuführen, um ein zuvor gesetztes Objekts freizugeben und dann die Instanzvariable auf `nil` zu setzen. All dies geschieht aber hinter den Kulissen. Für Sie als Programmierer sieht es einfach so aus, als weisen Sie der Variable `nil` zu.

```
self.colors = nil;
```

Senden Sie keine `release`-Nachrichten an beibehaltene Eigenschaften, etwa in der Form `[self.colors release]`. Dies wirkt sich nämlich nicht auf die Zuweisung der Instanzvariable `colors` aus, sodass die Variable danach auf einen wahrscheinlich freigegebenen Speicherbereich zeigt. Wenn Sie der beibehaltenen Eigenschaft später ein Objekt zuweisen, empfängt der Speicher, auf den `self.colors` zeigt, eine weitere `release`-Nachricht, was wahrscheinlich zu einem Absturz aufgrund einer Ausnahme wegen doppelter Freigabe führt.

3.6.3 Eigenschaften erstellen

Es gibt zwei grundlegende Arten von Eigenschaften, nämlich schreibbare (`readwrite`) und schreibgeschützte (`readonly`). Bei den schreibbaren Eigenschaften, die den Standardtyp darstellen, können Sie die Werte ändern, auf die Sie zugreifen, was bei den schreibgeschützten Eigenschaften nicht möglich ist. Sie brauchen auch zwei Arten von Zugriffsmethoden, nämlich Set- und Get-Methoden.

Set-Methoden legen Informationen fest, Get-Methoden rufen sie ab. Bei der Definition dieser Methoden können Sie willkürliche Namen wählen, sich aber auch an die üblichen Vereinbarungen in Objective-C halten: Für die Methode zum Abrufen des Objekts verwenden Sie den Namen der Instanzvariable, zum Setzen fügen Sie das Präfix `set` an. Objective-C kann diese Methoden sogar für Sie synthetisieren. Nehmen wir z. B. an, dass Sie in der Klassenschnittstelle von `Car` eine Eigenschaft wie `year` auf folgende Weise deklarieren:

```
@property int year;
```

In der Klassenimplementierung lassen Sie diese Eigenschaft dann wie folgt synthetisieren:

```
@synthesize year;
```

Anschließend können Sie die Instanzvariable lesen und setzen, ohne dafür weiteren Code schreiben zu müssen. Objective-C erstellt zwei Methoden, um den aktuellen Wert abzurufen (`[myCar year]`) und ihn zu setzen (`[myCar setYear:1962]`), und macht damit zusätzlich die beiden folgenden Kürzel in Punktschreibweise möglich:

```
myCar.year = 1962;
NSLog(@"%d", myCar.year);
```

Um eine schreibgeschützte Eigenschaft zu erstellen, müssen Sie sie in der Schnittstelle mit dem Attribut `readonly` deklarieren. Für schreibgeschützte Eigenschaften gibt es Get-, aber keine Set-Methoden. Die folgende Methode gibt z. B. einen formatierten Textstring mit Angaben über ein Auto zurück:

```
@property (readonly) NSString *carInfo;
```

Objective-C kann schreibgeschützte Eigenschaften synthetisieren, Sie können die Get-Methoden aber auch selbst erstellen und der Klassenimplementierung hinzufügen. Die folgende Methode gibt eine Beschreibung des Autos mithilfe von `stringWithFormat:` zurück, wobei ein Formatstring wie `sprintf` verwendet wird, um einen neuen String zu erstellen.

```
- (NSString *) carInfo
{
   if (!self.make) return @"";
   if (!self.model) return @"";
   return [NSString stringWithFormat:
      @"Car Info\nMake: %@nYear: % %d",
      self.make, self.model, self.year];
}
```

Diese Methode steht zur Verwendung in der Punktschreibweise zur Verfügung, z. B. als `CFShow(myCar.carInfo);`.

Wenn Sie die Get-Methode für eine schreibgeschützte Eigenschaft synthetisieren lassen, sollten Sie Vorsicht walten lassen. Achten Sie darauf, dass Sie die Instanzvariable für die Eigenschaft innerhalb der Implementierungsdatei nicht in der Punktschreibweise zuweisen. Wenn Sie z. B. `model` als schreibgeschützte Eigenschaft deklariert haben, können Sie sie wie folgt zuweisen:

```
model = @"Prefect";
```

Die folgende Schreibweise dagegen ist nicht möglich:

```
self.model = @"Prefect";
```

Bei dieser zweiten Version würde `setModel:` aufgerufen, aber eine solche Methode ist für eine schreibgeschützte Eigenschaft nicht definiert.

3.6.4 Eigene Get- und Set-Methoden erstellen

Wenn Sie Eigenschaften mit `@synthesize` synthetisieren lassen, erstellt Objective-C automatisch die Methoden dafür, doch können Sie diesen Vorgang auch übergehen und die Methoden selbst anlegen. Diese Methoden können so einfach aussehen wie die folgenden. Beachten Sie, dass das zweite Wort im Namen der Set-Methode mit großem Anfangsbuchstaben geschrieben wird. Vereinbarungsgemäß erwartet Objective-C, dass die Namen von Set-Methoden der Form `set`*`Instanz`*`:` folgen, wobei der erste Buchstabe des Namens der Instanzvariable großgeschrieben wird.

```
-(int) year
{
   return year;
}

- (void) setYear: (int) aYear
{
   year = aYear;
}
```

Wenn Sie Ihre eigenen Set- und Get-Methoden schreiben, können Sie dabei auch für die Speicherverwaltung sorgen. Die folgenden Methoden behalten Elemente bei und geben frühere Werte frei:

```
- (NSString *) model
{
   return model;
}

- (void) setModel: (NSString *) newModel
{
   if (newModel != model) {
      [model release];
      model = [newModel retain];
   }
}
```

Hinweis

In dem seltenen Fall, dass `newModel` auf irgendeine Weise ein Kind von `model` ist, kann der Aufruf `[model release]` unter Umständen auch den Speicher des `newModel`-Objekts freigeben. Aus diesem Grund sollten Sie in einer umfassenderen Set-Methode dafür sorgen, dass `newModel` beibehalten wird, bevor Sie `[model release]` aufrufen.

Sie können sogar noch weiter gehen und kompliziertere Routinen erstellen, die neben Zuweisung und Abruf noch zusätzlichen Nutzen mit sich bringen. Beispielweise können Sie zählen, wie oft ein Wert abgerufen oder geändert wurde, oder programminterne Nachrichten an andere Objekte senden. Dem Objective-C-Compiler ist das gleich, sofern er nur zu jeder Eigenschaft eine Get-Methode (gewöhnlich mit dem Namen der Eigenschaft) und eine Set-Methode (gewöhnlich `set`*`Eigenschaftsname`*) findet. Sie können sogar die Namenskonvention von Objective-C umgehen, indem Sie die Namen der Set- und Get-Methoden in der Eigenschaftsdeklaration angeben. Mit der folgenden Deklaration erstellen Sie die neue boolesche Eigenschaft `forSale` mit einem selbst definierten Paar aus Get- und Set-Methode. Eigenschaftsdeklarationen werden immer zur Klassenschnittstelle hinzugefügt.

```
@property (getter=isForSale, setter=setSalable:) BOOL forSale;
```

Synthetisieren Sie diese Methoden dann wie üblich in der Klassenimplementierung. Die Implementierung wird gewöhnlich in der `.m`-Datei gespeichert, die die `.h`-Headerdatei begleitet.

```
@synthesize forSale;
```

Mit diesem Ansatz erstellen Sie sowohl die normalen Set- und Get-Methoden mit Punktschreibweise als auch die beiden eigenen Methoden `isForSale` und `setSalable:`. Sie können `forSale` mit der Punktschreibweise zuweisen und abrufen, aber merkwürdigerweise nicht die entsprechenden Methoden einsetzen und auch nicht die eigene Set-Methode in der Punktschreibweise verwenden, wie Sie im Folgenden sehen:

```
Car *myCar = [Car car];

// Sie können die synthetisierten Set- und Get-Methoden verwenden
[myCar setSalable:YES];
printf("The car %s for sale\n",
   myCar.isForSale ? "is" : "is not");

// Die normalen Get- und Set-Methoden funktionieren auch in
// Punktschreibweise
myCar.forSale = NO;
printf("The car %s for sale\n",
   myCar.forSale ? "is" : "is not");

// Bei den Methodenversionen geht das dagegen nicht.
// Hierbei ergeben sich Laufzeitfehler.
// [myCar setForSale:YES];
// printf("The car %s for sale\n",
//    [myCar forSale] ? "is" : "is not");
// Sie können die selbst geschriebene Set-Methode nicht in der
// Punktschreibweise verwenden.
// Dies führt zu einem Kompilierungsfehler.
// myCar.setSalable = YES;
```

3.6.5 Eigenschaftsattribute

Neben den Attributen `readwrite` bzw. `readonly` können Sie auch angeben, ob eine Eigenschaft beibehalten werden soll (`retain`) und ob sie elementar ist (`atomic`). Das Standardverhalten für Eigenschaften ist die Zuweisung (`assign`). Mit einer solchen Eigenschaft ist kein besonderes Beibehaltungs- oder Freigabeverhalten verbunden, aber dadurch, dass Sie eine Variable zu einer Eigenschaft machen, sorgen Sie dafür, dass sie außerhalb der Klasse über die Punktschreibweise zugänglich wird. Eine folgendermaßen deklarierte Eigenschaft weist das Zuweisungsverhalten auf:

```
@property NSString *make;
```

Wenn Sie für die Eigenschaft das Attribut `retain` festlegen, hat das zweierlei Auswirkungen. Erstens wird das bei der Zuweisung übergebene Objekt beibehalten, zweitens wird der vorherige Wert freigegeben, bevor die neue Zuweisung erfolgt. Mit dem Attribut `retain` nutzen Sie die Vorteile der Speicherverwaltung, die wir im vorhergehenden Abschnitt behandelt haben. Um beibehaltene Eigenschaften zu erstellen, fügen Sie in der Deklaration das betreffende Attribut in Klammern ein:

```
@property (retain) NSString *make;
```

Ein drittes Attribut namens `copy` sendet eine Kopiernachricht an das übergebene Objekt, behält es bei und gibt einen eventuellen vorherigen Wert frei:

```
@property (copy) NSString *make;
```

Sie können das Objekt auch bei der Zuweisung beibehalten:

```
myCar.make = @"Ford";
[myCar.make retain];
```

Bei der Entwicklung in einer Multithread-Umgebung ist es sinnvoll, elementare Methoden zu verwenden. Xcode synthetisiert elementare Methoden, um Objekte automatisch zu sperren, bevor sie abgerufen oder verändert werden, und um sie anschließend wieder zu entsperren. Dadurch wird sichergestellt, dass der Vorgang zum Festlegen oder Abrufen eines Objektwerts unabhängig von gleichzeitig ablaufenden Threads vollständig ausgeführt wird. Es gibt jedoch kein Schlüsselwort `atomic`, denn alle Methoden werden automatisch als elementar synthetisiert. Sie können jedoch das Gegenteil verlangen, sodass Objective-C nicht elementare Zugriffsmethoden erstellt:

```
@property (nonatomic, retain) NSString *make;
```

Bei nicht elementaren Methoden erfolgt der Zugriff schneller. Es kann jedoch Probleme geben, wenn zwei konkurrierende Threads versuchen, gleichzeitig dieselbe Eigenschaft zu verändern. Elementare Eigenschaften stellen durch ihr Sperrverhalten sicher, dass ein Objekt vollständig aktualisiert wird, bevor die Eigenschaft für einen anderen Lese- oder Änderungsvorgang freigegeben wird.

3.7 Einfache Speicherverwaltung

Im Grunde gibt es bei der Speicherverwaltung nur zwei einfache Regeln. Der Beibehaltungszähler eines Objekts hat bei dessen Erstellung den Wert 1 und bei der Freigabe den Wert 0. Ihnen als Entwickler obliegt es, sich um die Beibehaltung des Objekts während seiner Lebensdauer zu kümmern. Stellen Sie sicher, dass es von Anfang bis Ende vorhanden ist, ohne vorzeitig freigegeben zu werden, und sorgen Sie dafür, dass es schließlich freigegeben wird, wenn es an der Zeit ist, das zu tun. Was die Sache etwas komplizierter macht, ist der Autorelease-Pool von Objective-C. Wenn einige Objekte automatisch und andere manuell freigegeben werden müssen, wie handhaben Sie dann am besten die Objekte? Die folgende einfache Anleitung gibt Ihnen eine Grundvorstellung davon, wie Sie die Speicherverwaltung erledigen.

3.7.1 Objekte erstellen

Jedes Mal, wenn Sie nach dem `alloc/init`-Muster ein Objekt erstellen, setzen Sie seinen Beibehaltungszähler auf 1. Dabei spielt es keine Rolle, welche Klasse Sie verwenden oder was für ein Objekt Sie anlegen.

```
id myObject = [[SomeObject alloc] init];
```

Wenn Sie bei Variablen mit lokalem Gültigkeitsbereich das Objekt nicht vor dem Ende einer Methode freigeben, tritt ein Speicherleck auf. Die Referenz auf den Speicher wird entfernt, aber der Speicher selbst bleibt zugewiesen. Auch der Beibehaltungszähler behält den Wert +1 bei.

```
- (void) leakyMethod
{
   // Hier tritt ein Speicherleck auf
   NSArray *array = [[NSArray alloc] init];
}
```

Die richtige Art und Weise, das `alloc/init`-Muster zu verwenden, besteht darin, das Objekt zu erstellen, zu verwenden und freizugeben. Durch die Freigabe wird der Beibehaltungszähler auf 0 gesetzt. Am Ende der Methode wird die Zuweisung des Objekts aufgehoben.

```
- (void) properMethod
{
   NSArray *array = [[NSArray alloc] init];
   // Hier wird das Array verwendet
   [array release];
}
```

Für Autorelease-Objekte ist bei Variablen mit lokalem Gültigkeitsbereich eine explizite `release`-Anweisung nicht erforderlich (tatsächlich dürfen Sie eine solche Anwendung gar nicht verwenden, da ihr Programm sonst aufgrund einer doppelten Freigabe abstürzt). Wenn Sie einem Objekt die Nachricht `autorelease` senden, wird es für die automatische Freigabe gekennzeichnet. Wird der Autorelease-Pool am Ende einer Ereignisschleife geleert, so sendet er an alle Objekte in seinem Besitz eine `release`-Nachricht.

```
- (void) anotherProperMethod
{
   NSArray *array = [[[NSArray alloc] init] autorelease];
   // Im Gegensatz zur Verwendung von release stürzt das Programm
   // hierbei nicht ab
   printf("Retain count is %d\n", [array retainCount]);
   // Hier wird das Array verwendet
}
```

Vereinbarungsgemäß geben alle Methoden zum Erstellen von Klassenobjekten Objekte mit automatischer Freigabe zurück. Die Klassenmethode `array` von `NSArray` gibt ein neu initialisiertes Array zurück, das bereits für die automatische Freigabe vorgesehen ist. Das Objekt kann innerhalb der Methode verwendet werden und wird freigegeben, wenn der Autorelease-Pool geleert wird.

```
- (void) yetAnotherProperMethod
{
   NSArray *array = [NSArray array];
   // Hier wird das Array verwendet
}
```

Am Ende dieser Methode kehrt das Autorelease-Array in den allgemeinen Speicherpool zurück.

3.7.2 Autorelease-Objekte erstellen

Wenn Sie eine andere Methode anweisen, ein Objekt zu erstellen, zeugt es von gutem Programmierstil, wenn dieses Objekt mit automatischer Freigabe zurückgegeben wird. Wenn Sie dies konsequent tun, können Sie sich auf die folgende Regel verlassen: »Wenn ich es nicht selbst zugewiesen habe, wurde es für mich erstellt und als Autorelease-Objekt zurückgegeben.«

```
- (Car *) fetchACar
{
   Car *myCar = [[Car alloc] init];
   return [myCar autorelease];
}
```

Dies gilt vor allem für Klassenmethoden. Vereinbarungsgemäß erstellen alle Klassenmethoden neue Objekte als Autorelease-Objekte. Im Allgemeinen spricht man hier von Hilfsmethoden. Objekte, die Sie selbst zuweisen, werden dagegen nicht für die automatische Freigabe gekennzeichnet, es sei denn, dass Sie dies ausdrücklich selbst verlangen.

```
// Wird nicht automatisch freigegeben
Car *car1 = [[Car alloc] init];

// Wird automatisch freigegeben
Car *car2 = [[[Car alloc] init] autorelease];

// Dies *sollte* vereinbarungsgemäß ein automatisch freigegebenes
// Objekt sein
Car *car3 = [Car car];
```

Um eine Klassen-Hilfsmethode zu erstellen, müssen Sie sie mit dem Präfix + statt - definieren und das Objekt zurückgeben, nachdem Sie ihm eine `autorelease`-Nachricht gesendet haben.

```
+ (Car *) car
{
   return [[[Car alloc] init] autorelease];
}
```

Lebensdauer von Autorelease-Objekten

Wie lange können Sie ein Autorelease-Objekt verwenden? Welche Garantien haben Sie? Die einfache Regel lautet, dass das Objekt so lange Ihnen gehört, bis das nächste Element der Ereignisschleife verarbeitet wird. Die Ereignisschleife wiederum wird dadurch ausgelöst, dass ein Benutzer etwas antippt oder eine Taste drückt, dass ein Ereignis aufgrund eines abgelaufenen Zeitraums eintritt usw. Diese Zeiträume sind nach menschlichem Maßstab unglaublich kurz, im Bezugsrahmen des iPhone-Prozessors aber ziemlich lang. Allgemeiner gesagt, können Sie davon ausgehen, dass ein Autorelease-Objekt während der Dauer eines Methodenaufrufs bestehen bleibt.

Nachdem eine Methode die Steuerung zurückgegeben hat, können Sie sich jedoch auf nichts mehr verlassen. Wenn Sie ein Array auch außerhalb des Gültigkeitsbereichs einer einzigen Methode oder für einen längeren Zeitraum benötigen (z. B. können Sie eine benutzerdefinierte Ausführungsschleife in einer Methode beginnen und verlängern, solange die Methode besteht), gelten andere Regeln. Dann müssen Sie Autorelease-Objekte beibehalten, um ihren Zählerstand zu erhöhen und zu verhindern, dass ihre Zuweisung beim Leeren des Pools aufgehoben wird. Wenn der Autorelease-Pool die Freigabe ihres Speichers verlangt, beläuft sich ihr Beibehaltungszähler dann immer noch auf mindestens 1.

Hinweis

Weisen Sie Eigenschaften nicht sich selbst zu, z. B. `myCar.colors = myCar.colors`. Das Verhalten der Eigenschaften nach dem Motto »erst freigeben, dann beibehalten« kann dazu führen, dass die Zuweisung dieser Objekte aufgehoben wird, bevor sie erneut zugewiesen und beibehalten werden können.

3.7.3 Autorelease-Objekte beibehalten

Sie können `retain` genauso an Autorelease- wie an jegliche anderen Objekte senden. Wenn Objekte, die für die automatische Freigabe gekennzeichnet sind, beibehalten werden, können sie über die Dauer einer einzelnen Methode hinaus fortbestehen. Nachdem ein Autorelease-Objekt beibehalten wurde, ist es ebenso anfällig für Speicherlecks wie ein Objekt, das Sie mit `alloc/init` erstellen. Beispielsweise kann ein Speicherleck auftreten, wenn Sie wie folgt ein Objekt mit dem Gültigkeitsbereich einer lokalen Variable beibehalten:

```
- (void)anotherLeakyMethod
{
   // Nach der Rückgabe der Steuerung verlieren Sie die lokale
   // Referenz auf das Array und können es nicht mehr freigeben.
   NSArray *array = [NSArray array];
   [array retain];
}
```

Wenn `array` erstellt wird, hat es einen Beibehaltungszähler von +1. Sobald Sie diesem Objekt eine `retain`-Nachricht senden, wird der Zählerwert auf +2 erhöht. Wenn die Methode endet und der Autorelease-Pool geleert wird, empfängt das Objekt eine einzige `release`-Nachricht, sodass der Zählerwert wieder auf 1 sinkt. Jetzt steckt das Objekt fest. Bei einem Zählerstand von +1 kann seine Zuweisung nicht aufgehoben werden, aber da keine Referenz mehr übrig ist, die auf das Objekt zeigt, kann es auch keine endgültige `release`-Nachricht empfangen, um seinen Lebenszyklus zu beenden. Aus diesem Grund ist es von entscheidender Bedeutung, Referenzen auf beibehaltene Objekte zu erstellen.

Durch eine Referenz können Sie ein beibehaltenes Objekt während seiner Lebensdauer verwenden und es freigeben, sobald Sie es nicht mehr brauchen. Referenzen setzen Sie über eine Instanzvariable (die bevorzugte Vorgehensweise) oder über eine statische Variable, die in der Klassenimplementierung definiert ist. Um den Code einfach und zuverlässig zu gestalten, verwenden Sie beibehaltene Eigenschaften, die aus diesen Instanzvariablen erstellt wurden. Im nächsten Abschnitt lernen Sie, wie beibehaltene Eigenschaften funktionieren und warum sie die bevorzugte Lösung für Entwickler sind.

3.7.4 Beibehaltene Eigenschaften

Beibehaltene Eigenschaften halten die Daten fest, die Sie ihnen zuweisen, und verwerfen sie, wenn Sie einen neuen Wert setzen. Daher fügen sie sich problemlos in die elementare Speicherverwaltung ein. Im Folgenden erfahren Sie, wie Sie beibehaltene Eigenschaften in Ihren iPhone-Anwendungen erstellen und verwenden.

Als Erstes deklarieren Sie die beibehaltene Eigenschaft in der Klassenschnittstelle, indem Sie das Schlüsselwort `retain` in Klammern angeben:

```
@property (retain) NSArray *colors;
```

Anschließend synthetisieren Sie in der Implementierung die Eigenschaftsmethoden:

```
@synthesize colors;
```

Ist die Direktive `@synthesize` vorhanden, erstellt Objective-C automatisch Routinen zur Handhabung der beibehaltenen Eigenschaft. Diese Routinen behalten ein Objekt automatisch bei, wenn Sie es zu einer Eigenschaft zuweisen. Dieses Verhalten gilt unabhängig davon, ob das Objekt für die automatische Freigabe eingerichtet ist oder nicht. Wenn Sie der Eigenschaft ein anderes Objekt zuweisen, wird das vorherige automatisch freigegeben.

Werte zu beibehaltenen Eigenschaften zuweisen

Bei der Arbeit mit beibehaltenen Eigenschaften müssen Sie sich darüber im Klaren sein, dass es zwei Formen der Zuweisung gibt. Welchem Muster Sie folgen, hängt davon ob, ob Sie ein Autorelease-Objekt zuweisen oder ein normales. Bei Objekten mit automatischer Freigabe verwenden Sie eine einfache Zuweisung. Das folgende Beispiel zeigt eine Zuweisung, die die Eigenschaft `colors` auf das neue Array setzt und dieses beibehält:

```
myCar.colors = [NSArray arrayWithObjects:
   @"Black", @"Silver", @"Gray", nil];
```

Das Array wird erstellt und als Autorelease-Objekt mit dem Beibehaltungszähler +1 zurückgegeben. Durch die Zuweisung zur beibehaltenen Eigenschaft `colors` wird der Zählerwert auf +2 heraufgesetzt. Am Ende der aktuellen Ereignisschleife sendet der Autorelease-Pool eine `release`-Nachricht an das Array, sodass der Zähler auf +1 zurückfällt.

Normale Objekte (ohne automatische Freigabe) geben Sie nach der Zuweisung frei. Wird ein normal zugewiesenes Objekt erstellt, beträgt sein Beibehaltungszähler +1. Bei der Zuweisung zu einer beibehaltenen Eigenschaft steigt dieser Wert auf +2. Die Freigabe des Objekts setzt den Zähler wieder auf +1 zurück.

```
// Nicht automatisch freigegebenes Objekt. Der Beibehaltungszähler
// beläuft sich bei der Erstellung auf +1
NSArray *array = [[NSArray alloc]
   initWithObjects:@"Black", @"Silver", @"Gray", nil];

// Zählerstand steigt bei der Zuweisung zu einer beibehaltenen Eigenschaft
// auf +2
myCar.colors = array;

// Durch die Freigabe wird der Zähler wieder auf +1 gesenkt
[array release];
```

Dem Muster »Erstellen – Zuweisen – Freigeben« begegnen Sie bei der iPhone-Entwicklung häufiger. Sie können es verwenden, wenn Sie eine neu im Speicher zugewiesene Ansicht zu einem Ansichtscontrollerobjekt zuweisen, z. B. wie folgt:

```
UIView *mainView = [[UIView alloc] initWithFrame:aFrame];
self.view = mainView;
[mainView release];
```

In diesen drei Schritten wird der Beibehaltungszähler des Objekts von +1 auf +2 erhöht und dann wieder auf +1 gesenkt.

Ein Zählerendstand von +1 sorgt dafür, dass Sie das Objekt beliebig lange verwenden können. Gleichzeitig ist auch sichergestellt, dass das Objekt freigegeben wird, wenn die Eigenschaft auf einen neuen Wert gesetzt und für den früheren Wert `release` aufgerufen wird. Bei dieser Freigabe wird der Zähler von +1 auf 0 gesetzt, sodass die Zuweisung des Objekts automatisch aufgehoben wird.

Beibehaltene Eigenschaften erneut zuweisen

Wenn Sie eine beibehaltene Eigenschaft nicht mehr brauchen, setzen Sie sie (unabhängig davon, wie Sie sie erstellt haben) auf `nil` oder ein anderes Objekt. Dadurch wird dem zuvor zugewiesenen Objekt eine `release`-Nachricht gesendet.

```
myCar.colors=nil;
```

Haben Sie die Eigenschaft `colors` wie in unserem Beispiel auf ein Array gesetzt, erhält dieses Array automatisch eine `release`-Nachricht. Da bei jeder Zuweisung der Beibehaltungszähler des Objekts auf +1 steigt, setzt die Neuzuweisung ihn wieder von +1 auf 0 zurück. Damit endet die Lebensdauer des Objekts.

Fallgruben bei der Zuweisung vermeiden

Innerhalb der Klassenimplementierung erweist sich die Verwendung von Eigenschaften als praktisch, da Sie damit die Möglichkeiten der Speicherverwaltung nutzen können. Vermeiden Sie dabei die unmittelbare Verwendung von Instanzvariablen, da bei einer solchen direkten Zuweisung weder das Array beibehalten noch ein früherer Wert freigegeben wird. Dies ist eine Fallgrube, in die viele Anfänger stürzen. Denken Sie beim Zugriff auf Instanzvariablen auch an die Punktschreibweise.

```
colors = [NSArray arrayWithObjects:
   @"Black", @"Silver", @"Gray", nil];
```

Dieselben Vorsichtsmaßnahmen gelten auch für Eigenschaften, die als `assign` definiert sind. Sehen Sie sich die folgenden Verhaltsweisen genau an. Sowohl bei

```
@property NSArray *colors;
```

als auch bei

```
@property NSArray (assign) *colors;
```

können Sie die Punktschreibweise verwenden, doch bei der Zuweisung über diese Eigenschaften werden keinerlei Objekte beibehalten oder freigegeben. Mit `assign` definierte Eigenschaften machen die Instanzvariable `colors` zwar von außen zugänglich, bieten aber nicht die Vorteile der Speicherverwaltung, die sich bei `retain`-Eigenschaften ergeben.

Hinweis

Als Faustregel rät Apple davon ab, Eigenschaften in `init`-Funktionen zu verwenden. Stattdessen sollten Sie dort direkt Instanzvariablen einsetzen.

3.7.5 Hohe Werte für den Beibehaltungszähler

Wenn der Wert für den Beibehaltungszähler über +1 hinausgeht und dort auch bleibt, heißt das nicht unbedingt, dass Sie etwas falsch gemacht haben. Betrachten Sie das folgende Codefragment, in dem eine Sicht erstellt und zu verschiedenen Arrays hinzugefügt wird. Der Beibehaltungszähler steigt dabei von +1 auf +4.

```
// Wenn die Ansicht erstellt wird, beträgt ihr Beibehaltungszähler +1
UIView *view = [[[UIView alloc] init] autorelease];
printf("Count: %d\n", [view retainCount]);

// Wenn Sie die Ansicht zu einem Array hinzufügen, erhöht sich der Wert
// auf +2
NSArray *array1 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);

// Bei einem weiteren Array steigt der Beibehaltungszähler auf +3
NSArray *array2 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);

// Noch ein Array führt zum Wert +4
NSArray *array3 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);
```

Jedes Array wurde mit einer Klassen-Hilfsmethode erstellt und gibt ein Autorelease-Objekt zurück. Auch die Ansicht ist für die automatische Freigabe gekennzeichnet. Manche Sammlungsklassen wie `NSArray` behalten Objekte automatisch bei, wenn sie sie zu einem Array hinzufügen, und geben sie frei, wenn das Objekt entfernt wird (dies gilt nur für veränderbare Objekte) oder wenn die Sammlung freigegeben wird. Dieser Code weist keine Speicherlecks auf, da jedes der vier Objekte so eingerichtet ist, dass es sich selbst und seine Kindobjekte korrekt freigibt, wenn der Autorelease-Pool geleert wird.

Wenn `release` zu den drei Arrays gesendet wird, gibt jedes von ihnen die Ansicht frei, was deren Beibehaltungswert von +4 auf +1 senkt. Die letzte `release`-Nachricht, die an das Objekt selbst geht, bringt den Zähler von +1 auf 0 herunter, sodass die Zuweisung der Ansicht am Ende der Methode aufgehoben werden kann. Kein Leck, keine weitere Beibehaltung, keine Probleme!

3.7.6 Andere Möglichkeiten zum Erstellen von Objekten

Sie haben gesehen, wie Sie mit `alloc` Speicher zuweisen. In Objective-C gibt es jedoch auch andere Möglichkeiten, um neue Objekte zu erstellen. Die Methoden dafür unterscheiden sich bei den einzelnen Klassen und Frameworks, weshalb Sie sich die Klassendokumentation ansehen müssen, um genauere Einzelheiten zu erfahren. Wenn Sie ein Objekt mit einer Methode erstellen, in deren Namen `alloc`, `new`, `create` oder `copy` vorkommt, haben Sie immer die Verantwortung dafür, das Objekt freizugeben. Im Gegensatz zu Klassen-Hilfsmethoden geben Methoden mit solchen Namen im Allgemeinen keine Autorelease-Objekte zurück.

Wenn Sie z. B. eine `copy`-Nachricht an ein Objekt senden, duplizieren Sie es. Dabei wird ein Objekt mit einem Beibehaltungswert von +1, aber ohne Zuweisung zum Autorelease-Pool zurückgegeben. Verwenden Sie `copy`, wenn Sie ein Objekt duplizieren und Änderungen an der Kopie vornehmen, das Original aber aufbewahren wollen. Beachten Sie, dass Objective-C von Sammlungen wie Arrays und Dictionarys meistens flache Kopien erstellt. Dabei werden zwar die Struktur der Sammlung und die Adressen der einzelnen Zeiger übernommen, die gespeicherten Inhalte aber nicht kopiert.

Objektzuweisung im C-Stil

Da Objective-C eine Obermenge von C ist, werden in Objective-C-Programmen für das iPhone häufig APIs verwendet, in denen Objekte wie in C erstellt und verwaltet werden. *Core Foundation* (CF) ist ein Cocoa Touch-Framework mit Funktionsaufrufen auf C-Basis. Wenn Sie in Objective-C mit CF-Objekten arbeiten, erstellen Sie sie mit `CFAllocators` und verwenden häufig die Funktion `CFRelease()`, um den Objektspeicher freizugeben.

Es gibt jedoch keine einfachen Regeln. Wie der folgende Code zeigt, ist es möglich, dass Sie `free()`, `CFRelease()` und eigene Methoden wie `CGContextRelease()` alle im selben Gültigkeitsbereich neben den standardmäßigen Klassen-Hilfsmethoden von Objective-C wie `imageWithCGImage:` verwenden. Um das Kontextobjekt zu erstellen, haben wir hier die Funktion `CGBitmaßContextCreate()` verwendet, die wie die meisten CF-Funktionsaufrufe kein Autorelease-Objekt zurückgibt. Das folgende Codefragment erstellt ein Objekt der iPhone-Klasse `UIImage`, das zum Speichern von Bilddaten dient:

```
UIImage *buildImage(int imgsize)
{
   // Erstellt Kontext mit zugewiesenen Bits
   CGContextRef context =
      MyCreateBitmapContext(imgsize, imgsize);
   CGImageRef myRef =
      CGBitmapContextCreateImage(context);
   free(CGBitmapContextGetData(context));
   CGContextRelease(context);
   UIImage *img = [UIImage imageWithCGImage:myRef];
   CFRelease(myRef);
   return img;
}
```

Carbon und Core Foundation

Da Sie noch oft genug mit Core Foundation arbeiten müssen, sollten Sie darauf vorbereitet sein, die darin enthaltenen Konstrukte zu verwenden, vor allem die Frameworks. Dabei handelt es sich um Bibliotheken mit Klassen, die Sie in Ihren Anwendungen einsetzen können.

Tabelle 3.2 erläutert die wichtigsten Begriffe. Zusammenfassend sei gesagt, dass in den ersten Versionen von Mac OS X das auf C aufgebaute Framework Core Foundation als Übergangssystem für die Entwicklung von Anwendungen verwendet wurde, die sowohl auf klassischen Mac-Systemen als auch auf Mac OS X laufen konnten. Core Foundation verwendet zwar objektorientierte Erweiterungen zu C, die Funktionen und Konstrukte basieren aber auf C, nicht auf Objective-C.

Begriff	Erklärung
Foundation	Die Kernklassen für die Objective-C-Entwicklung, die alle grundlegenden Datentypen und Dienste bereitstellen, die für Cocoa und Cocoa Touch erforderlich sind. In einem Abschnitt am Ende dieses Kapitels erhalten Sie eine Einführung in einige der wichtigsten Foundation-Klassen für Ihre Anwendungen.
Core Foundation	Eine Bibliothek mit C-Klassen, die zwar auf der Foundation-API beruhen, aber in C implementiert sind. Core Foundation verwendet objektorientierte Daten, ist aber nicht aus Objective-C-Klassen aufgebaut.
Carbon	Ein älterer Satz von Apple bereitgestellter Bibliotheken mit einer prozeduralen API. Carbon bot Unterstützung für Ereignisbehandlung, eine Grafikbibliothek und viele weitere Frameworks. Manche Carbon-APIs bestehen in Core Foundation fort. Carbon wurde für das klassische Mac OS eingeführt und erschien zum ersten Mal in Mac OS 8.1.
Cocoa	Die von Apple bereitgestellte Sammlung von Frameworks, APIs und Laufzeitumgebungen, die das moderne Laufzeitsystem von Mac OS X bilden. Die Frameworks sind größtenteils in Objective-C geschrieben, obwohl in einigen weiterhin C bzw. C++ verwendet wird.
Cocoa Touch	Das Äquivalent zu Cocoa für iPhone OS, bei dem die Frameworks für die berührungsempfindliche Benutzeroberfläche des iPhone angepasst sind. Einige iPhone-Frameworks wie Core Audio und Open GL ES werden nicht als Bestandteil von Cocoa Touch betrachtet.
Toll Free Bridging	Eine Methode zur Kombination von Cocoa und Carbon. Hierbei geht es um einen Satz von austauschbaren Datentypen. Beispielsweise können das Foundation-Objekt `(NSString *)` von Cocoa und das Core Foundation-Objekt `CFStringRef` von Carbon gleichwertig verwendet werden. Bei diesem »zollfreien Brückenschlag« wird die auf C aufgebaute Core Foundation mit der Welt von Objective-C-Foundation verbunden.

▶ *Tabelle 3.2: Schlüsselbegriffe für die OS X-Entwicklung*

Die Core Foundation-Technologie besteht in Cocoa fort. Wenn Sie iPhone-Anwendungen in Objective-C programmieren, wird Ihnen irgendwann die C-artige Core Foundation begegnen. Die Besonderheiten der Core Foundation-Programmierung sind jedoch nicht Thema dieses Kapitels. Am besten lernen Sie sie kennen, indem Sie sie für sich betrachten und nicht, während Sie sich mit den Grundlagen der Programmierung in Objective-C beschäftigen.

3.7.7 Zuweisung von Objekten aufheben

Auf dem iPhone wird Objective-C mit einer Speicherverwaltung durch Referenzzähler verwendet. Es gibt auf dem iPhone keine Garbage Collection, und die Wahrscheinlichkeit dafür, dass es sie jemals geben wird, ist sehr gering. Jedes Objekt räumt hinter sich selbst auf. Was heißt das aber für die Praxis? Die folgende kurze Zusammenfassung zeigt, wie Sie die Existenz eines Objekts beenden, wie Sie seine Instanzvariablen aufräumen und die Aufhebung der Zuweisung vorbereiten.

Instanzvariablen müssen beibehaltene Objekte freigeben, bevor die Zuweisung aufgehoben wird. Dabei müssen Sie als Entwickler dafür sorgen, dass der Beibehaltungswert dieser Objekte auf 0 sinkt, bevor das Elternobjekt freigegeben wird. Dazu implementieren Sie die Methode `dealloc`, die das Laufzeitsystem automatisch aufruft, wenn ein Objekt freigegeben werden soll. Wenn Sie eine Klasse verwenden, die Objektinstanzvariablen aufweist (also nicht nur Fließkommazahlen, Integer und boolesche Werte), müssen Sie wahrscheinlich eine Methode zum Aufheben der Zuweisung implementieren. Die Grundstruktur einer `dealloc`-Methode sieht wie folgt aus:

```
- (void) dealloc
{
    // Aufräumvorgang durch die Klasse
    Hier räume ich meine eigenen Instanzvariablen auf

    // Aufräumvorgang der Oberklasse
    [super dealloc]
}
```

Die Methode, die Sie schreiben, sollte zwei Phasen aufweisen. Als Erstes werden jegliche Instanzvariablen Ihrer Klasse aufgeräumt, anschließend weisen Sie die Oberklasse an, ihre eigene Aufräumroutine durchzuführen. Das Schlüsselwort `super` verweist auf die Oberklasse des Objekts, das die `dealloc`-Methode ausführt. Wie Sie den Aufräumvorgang gestalten, hängt davon ab, ob die Instanzvariablen automatisch beibehalten werden.

Sie haben bereits gelesen, wie Sie Objekte erstellen, wie Sie Referenzen auf Objekte anlegen und wie Sie sicherstellen, dass der Beibehaltungswert eines Objekts nach dessen Erstellung bei +1 bleibt. Jetzt sehen wir uns den letzten Schritt in der Existenz eines Objekts an, nämlich die Verringerung des Zählerwerts auf 0, damit die Zuweisung des Objekts aufgehoben werden kann.

Beibehaltene Eigenschaften

Beibehaltene Eigenschaften müssen Sie mit einer Zuweisung in Punktschreibweise auf `nil` setzen. Dadurch wird die von Objective-C synthetisierte benutzerdefinierte Set-Methode aufgerufen und das Objekt freigegeben, das der Eigenschaft zuvor zugewiesen war. Hatte dieses frühere Objekt einen Beibehaltungswert von +1, wird der Zähler durch diese Freigabe auf 0 gesetzt.

```
self.make = nil;
```

Variablen

Wenn Sie einfache Instanzvariablen (keine Eigenschaften) verwenden oder Formateigenschaften zuweisen, senden Sie die Nachricht `release`, wenn die Zuweisung aufgehoben werden soll. Nehmen wir an, Sie haben die Instanzvariable `salesman` definiert. Sie kann zu einem beliebigen Zeitpunkt während der Lebensdauer des Objekts gesetzt werden. Eine Zuweisung von `salesman` kann wie folgt aussehen:

```
// Freigabe des früheren Werts
[salesman release];

// Neue Zuweisung erfolgt. Beibehaltungswert beträgt +1
salesman = [[SomeClass alloc] init];
```

Bei einer Zuweisung dieser Art kann `salesman` zu jedem Zeitpunkt der Lebensdauer des Objekts auf ein Objekt mit einem Beibehaltungswert von +1 zeigen. In Ihrer `dealloc`-Methode müssen Sie daher das Objekt, das zurzeit zu `salesman` zugewiesen ist, freigeben, um den Zählerwert auf 0 zu verringern.

```
[salesman release];
```

Beispiel für eine dealloc-Methode

Betrachten wir als Beispiel unsere erweiterte Klasse `Car` mit beibehaltenen Eigenschaften für `make`, `model` und `colors` sowie der einfachen Instanzvariable `salesman`. Die endgültige `dealloc`-Methode sieht dann wie im Folgenden gezeigt aus. Die Instanzvariablen `year` und `forSale` sind Integer- bzw. boolesche Werte und keine Objekte, weshalb sie nicht auf diese Weise verwaltet werden müssen.

```
- (void) dealloc
{
   self.make = nil;
   self.model = nil;
   self.colors = nil;
   [salesman release];
   [super dealloc];
}
```

Eine Obergrenze für den Beibehaltungswert ist entscheidend dafür, dass die Speicherverwaltung von Objective-C funktioniert. Kein Objekt sollte einen Beibehaltungswert größer als +1 haben, nachdem es erstellt und zugewiesen wurde. Die Grenze von +1 stellt sicher, dass die endgültige Freigabe in `dealloc` den Wert auf 0 herabsetzt.

Zusätzliche Aufräumvorgänge

Die Methode `dealloc` ist der ideale Ort, um aufzuräumen. Beispielsweise kann es sein, dass Sie einen Audio Toolbox-Klang loswerden oder andere Wartungsaufgaben verrichten müssen, bevor eine Klasse freigegeben werden kann. Solche Aufgaben betreffen fast ausnahmslos ältere Frameworks wie Core Foundation, Core Graphics und Core Audio oder ähnliche Frameworks im C-Stil.

```
if (snd) AudioServicesDisposeSystemSoundID(snd);
```

Betrachten Sie `dealloc` als Ihre letzte Gelegenheit, um lose Enden abzuschneiden, bevor das Objekt für immer verschwindet. Ob Sie nun offene Sockets herunterfahren, Dateizeiger schließen oder Ressourcen freigeben müssen – nutzen Sie diese Methode, um sicherzustellen, dass der Code in einen so sauberen Zustand wie möglich zurückkehrt.

3.8 Singletons verwenden

Über die Klassen `UIApplication` und `UIDevice` haben Sie Zugriff auf die zurzeit laufende Anwendung und die Gerätehardware, auf der sie ausgeführt wird. Zu diesem Zweck bieten diese Klassen Singletons an, also eine einzige Instanz einer Klasse im aktuellen Prozess. Beispielsweise gibt `[UIApplication sharedApplication]` ein Singleton zurück, das Informationen darüber zurückliefert, welcher Delegate verwendet wird, ob die Anwendung die Bearbeitung durch Schütteln unterstützt, welche Fenster das Programm definiert usw.

Die meisten Singleton-Objekte sind Steuerungszentralen. Unter anderem koordinieren sie Dienste, stellen Schlüsselinformationen bereit und lenken den externen Zugriff. Wenn Sie eine zentrale Funktion benötigen, z. B. einen Manager, der auf einen Webdienst zugreift, können Sie durch die Verwendung eines Singletons sicherstellen, dass alle Teile der Anwendung mit demselben zentralen Manager koordiniert werden.

Zum Erstellen eines Singletons benötigen Sie nur sehr wenig Code. Sie definieren eine statische, gemeinsam genutzte Instanz innerhalb der Klassenimplementierung und führen eine Klassenmethode hinzu, die auf diese Instanz zeigt. In dem folgenden Fragment, das aus dem Tagging-Beispiel von Kapitel 6, *Ansichten und Animationen zusammenstellen*, stammt, wird die Instanz erstellt, sobald sie zum ersten Mal angefordert wird.

```
@implementation ViewIndexer
static ViewIndexer *sharedInstance = nil;

+(ViewIndexer *) sharedInstance {
   if(!sharedInstance)
      sharedInstance = [[self alloc] init];
   return sharedInstance;
}

// Hier ist das Verhalten der Klasse definiert

@end
```

Um dieses Singleton zu verwenden, rufen Sie `[ViewIndexer sharedInstance]` auf. Dadurch wird das gemeinsam genutzte Objekt zurückgegeben, sodass Sie Zugriff auf das vom Singleton gebotene Verhalten haben. Um andere Klassen daran zu hindern, eine zweite Instanz zu erstellen, überschreiben Sie `allocWithZone:`. (Für die meisten Verwendungszwecke ist das jedoch schon so übervorsichtig, dass es ans Paranoide grenzt.) Die hier verwendete Direktive `@synchronized()` verhindert, dass dieser Code durch mehr als einen Thread auf einmal ausgeführt wird.

```
+ (id)allocWithZone:(NSZone *)zone
{
   @synchronized(self) {
      if (sharedInstance == nil) {
         sharedInstance = [super allocWithZone:zone];
         return sharedInstance;
      }
   }
   return nil;
}
```

3.9 Kategorien (zur Erweiterung von Klassen)

Die Möglichkeit, bereits bestehende Klassen zu erweitern, ist eines der leistungsstärksten Merkmale von Objective-C. Die Erweiterung des Verhaltens wird als *Kategorie* bezeichnet. Kategorien erweitern den Funktionsumfang von Klassen, ohne dass Sie Unterklassen bilden müssen. Stattdessen wählen Sie einen beschreibenden Namen für die Erweiterung aus, erstellen einen Header und implementieren die Funktionalität in einer Methodendatei. Kategorien können Methoden auch zu bestehenden Klassen hinzufügen, die beim Erstellen der Kategorie noch nicht definiert waren und deren Quellcode nicht vorlag.

Um eine Kategorie zu erstellen, müssen Sie eine neue Schnittstelle deklarieren. Geben Sie den Namen der Kategorie (Sie können ihn beliebig wählen) in Klammern an, wie Sie im folgenden Beispiel sehen. Führen Sie dann alle neuen öffentlichen Methoden und Eigenschaften auf, und speichern Sie die Headerdatei. Die Kategorie `Orientation` im Beispiel erweitert die Klasse `UIDevice`, bei der es sich um die SDK-Klasse für die Meldung von Gerätemerkmalen wie Ausrichtung, Ladezustand der Batterie und Status des Annäherungssensors handelt. Diese Schnittstelle fügt `UIDevice` eine einzige Eigenschaft hinzu, die einen schreibgeschützten booleschen Wert zurückgibt. Die neue Eigenschaft `isLandscape` meldet, ob das Gerät zurzeit quer ausgerichtet ist.

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end
```

Anders als beim Erstellen von Unterklassen können Sie in einer Kategorieschnittstelle keine neuen Instanzvariablen hinzufügen. Stattdessen erweitern Sie das Verhalten der Klasse, wie der Quellcode von *Listing 3.3* zeigt. Hier wird die Überprüfung der Ausrichtung durch einen Blick auf die Standardeigenschaft `orientation` von `UIDevice` implementiert.

Diese neue Eigenschaft können Sie wie folgt verwenden:

```
NSLog(@"The device orientation is%@landscape",
   [UIDevice currentDevice].isLandscape ? @" " : @" not ");
```

Hier ist die Prüfung der Querausrichtung nahtlos in die SDK-Klasse `UIDevice` eingebaut, und zwar über eine Eigenschaft, die es vor der Erweiterung der Klasse nicht gab. Nebenbei bemerkt, enthält `UIKit` Makros zum Thema Geräteausrichtung (`UIDeviceOrientationPortrait` und `UIDeviceOrientationLandscape`), denen Sie aber einen vom Gerät abgerufenen Ausrichtungswert übergeben müssen.

Hinweis

Mithilfe von Kategorien können Sie nicht nur neues Verhalten zu bestehenden Klassen hinzufügen, sondern auch verwandte Methoden für selbst erstellte Klassen in getrennten Dateien gruppieren. Bei umfangreichen, komplexen Klassen kann dies die Wartung und die Verwaltung der einzelnen Quelldateien erleichtern. Wenn Sie eine Kategoriemethode hinzufügen, die die gleiche Signatur hat wie eine bestehende Methode, verwendet das Laufzeitsystem von Objective-C Ihre Implementierung und überschreibt das Original.

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end

@implementation UIDevice (Orientation)
- (BOOL) isLandscape
{
   return (self.orientation == UIDeviceOrientationLandscapeLeft) ||
      (self.orientation == UIDeviceOrientationLandscapeRight);
}
@end
```

▶ *Listing 3.3: Die Kategorie* `Orientation` *für die Klasse* `UIDevice` *erstellen*

3.10 Protokolle

In Kapitel 1 wurden Delegates eingeführt, mit deren Hilfe sich Einzelheiten implementieren lassen, die während der Definition der Klasse noch nicht bekannt sind. Beispielsweise ist einer Tabelle zwar klar, wie die Zellenreihen angezeigt werden sollen, sie weiß aber nicht, was zu geschehen hat, wenn der Benutzer auf eine Zelle tippt. Die Bedeutung einer solchen Berührung hängt von der Anwendung ab, die die Tabelle implementiert. Beim Antippen kann ein anderer Bildschirm geöffnet, eine Nachricht an einen Webserver gesendet oder irgendeine andere denkbare Aufgabe ausgeführt werden. Durch

Delegierung kann die Tabelle mit einem intelligenten Objekt kommunizieren, das für die Handhabung solcher Berührungen zuständig ist, dessen Verhalten aber nicht beim Erstellen der Tabellenklasse geschrieben wird, sondern zu einem davon völlig unabhängigen Zeitpunkt.

Die Delegierung stellt im Grunde eine Sprache für die Vermittlung zwischen einem Objekt und seinem Handler bereit. Eine Tabelle teilt ihrem Delegate mit, dass sie angetippt, dass in ihr geblättert oder dass ihr Status auf andere Weise verändert wurde. Der Delegate entscheidet dann, wie auf diese Nachrichten zu reagieren ist, und führt auf der Grundlage der Semantik seiner Anwendung die entsprechenden Aktualisierungen herbei.

Datenquellen funktionieren ähnlich, doch vermitteln Sie keine Reaktionen auf Vorgänge, sondern stellen Daten auf Anforderung bereit. Wenn eine Tabelle fragt, welche Informationen sie in Zelle 1 und 2 packen soll, antwortete die Datenquelle mit den angeforderten Informationen. Wie bei der Delegierung können Tabellen auch bei Datenquellen Anfragen an ein Objekt richten, das die jeweiligen Anforderungen kennt.

In Objective-C werden sowohl die Delegierung als auch die Verwendung von Datenquellen durch das System der sogenannten *Protokolle* ermöglicht. Protokolle definieren im Voraus, wie Klassen miteinander kommunizieren können. Sie enthalten eine Liste von Methoden, die nicht innerhalb von Klassen definiert sind. Einige dieser Methoden sind erforderlich, andere optional. Wenn eine Klasse die erforderlichen Methoden implementiert, so spricht man davon, dass sie dem Protokoll gehorcht oder es erfüllt.

3.10.1 Ein Protokoll definieren

Nehmen wir als Beispiel einen Springteufel – einen kleinen Kasten mit einer Kurbel. Wenn Sie die Kurbel drehen, spielt Musik, und manchmal springt eine Puppe aus dem Kasten. Stellen Sie sich jetzt vor, dass Sie dieses Spielzeug (besser gesagt, eine grobe Annäherung) in Objective-C implementieren. Es gibt eine mögliche Aktion, nämlich das Drehen der Kurbel, und zwei mögliche Ergebnisse: Musik und das Erscheinen des Teufelchens.

Betrachten wir nun einen Programmclient für dieses Spielzeug. Er könnte beispielsweise auf die Ergebnisse reagieren, indem er beim Abspielen der Musik nach und nach einen Langeweilezähler erhöht und beim Erscheinen des Teufelchens Überraschung zeigt. In Objective-C müssen Sie für diesen Client also zwei Reaktionen implementieren, einen für die Musik und einen für das Teufelchen (im Englischen »jack« genannt). Ein mögliches Clientprotokoll sieht wie folgt aus:

```
@protocol JackClient
- (void) musicDidPlay;
- (void) jackDidAppear;
@end
```

Dieses Protokoll legt fest, dass etwas auf das Abspielen der Musik und das Erscheinen des Springteufels reagieren muss, um ein Client des Spielzeugs zu sein. Die Auflistung dieser Methoden innerhalb eines `@protocol`-Containers definiert das Protokoll. Alle hier aufgeführten Methoden sind erforderlich, sofern Sie sie nicht ausdrücklich als `@optional` deklarieren. Mehr darüber erfahren Sie im nächsten Abschnitt.

3.10.2 Ein Protokoll eingliedern

Stellen Sie sich als Nächstes vor, dass Sie eine Klasse für das Spielzeug selbst entwerfen. Sie stellt eine mögliche Aktion bereit, nämlich das Drehen der Kurbel, und erfordert ein zweites Objekt, das das Protokoll implementiert und in diesem Fall als Client bezeichnet wird. Die Schnittstelle der Klasse legt fest, dass der Client eine Art von Objekt (`id`) sein muss, das dem Protokoll `JackClient` gehorcht (`id <JackClient>`). Darüber hinaus weiß die Klasse zu diesem Zeitpunkt nichts darüber, was für eine Art von Objekt dieser Dienst bereitstellt.

```
@interface JackInTheBox : NSObject
{
   id <JackClient> client;
}
- (void) turnTheCrank;
@property (retain) id <JackClient> client;
@end
```

3.10.3 Callbacks hinzufügen

Über Callbacks wird die Klasse für das Spielzeug mit ihrem Client verbunden. Da der Client das Protokoll `JackClient` erfüllen muss, können Sie ihm die Nachrichten `jackDidAppear` und `musicDidPlay` senden, ohne dass es bei der Kompilierung zu einem Fehler kommt. Das Protokoll sorgt dafür, dass der Client diese Methoden implementiert. Im folgenden Code wird die Callback-Methode zufällig ausgewählt. Bei ungefähr neun von zehn Aufrufen spielt die Musik, sodass `musicDidPlay` an den Client gesendet wird.

```
- (void) turnTheCrank
{
   // Sie brauchen einen Client, der auf das Drehen der Kurbel reagiert
   if (!self.client) return;

   // Zufällig ausgewählte Reaktion auf die Kurbeldrehung
   int action = random() % 10;
   if (action < 1)
      [self.client jackDidAppear];
   else
      [self.client musicDidPlay];
}
```

3.10.4 Optionale Callbacks deklarieren

In Protokollen treten zwei Arten von Callbacks auf, nämlich erforderliche und optionale, wobei erforderliche Callbacks den Standard bilden. Eine Klasse, die dem Protokoll gehorcht, muss diese Methode implementieren, da ansonsten eine Compiler-Warnung erfolgt. Mit den Schlüsselwörtern `@required` und `@optional` können Sie festlegen, zu welcher Art eine Methode gehört. Alle hinter

`@required` aufgeführten Methoden sind erforderlich, alle hinter `@optional` aufgeführten sind optional. Daraus können Sie Ihr Protokoll nach Bedarf zusammensetzen.

```
@protocol JackClient <NSObject>
- (void) musicDidPlay; // erforderlich
@required
- (void) jackDidAppear; // ebenfalls erforderlich
@optional
- (void) nothingDidHappen; // optional
@end
```

In der Praxis ist es unsinnig, wenn mehr Schlüsselwörter erscheinen als ein einziges Mal `@optional`. Das gleiche Protokoll können Sie auch einfacher deklarieren. Wenn Sie gar keine optionalen Elemente verwenden, lassen Sie das Schlüsselwort ganz weg. Beachten Sie die Deklaration `<NSObject>` in dem Beispiel. Sie ist erforderlich, damit optionale Protokolle implementiert werden können. Diese Deklaration besagt, dass ein `JackClient`-Objekt eine Art von `NSObject` ist.

```
@protocol JackClient <NSObject>
- (void) musicDidPlay;
- (void) jackDidAppear;
@optional
- (void) nothingDidHappen;
@end
```

3.10.5 Optionale Callbacks implementieren

Bei optionalen Protokollmethoden hat der Client die Wahl, ob er sie implementieren will. Dadurch verringern Sie die Implementierungspflichten für denjenigen, der den Client schreibt, laden der Klasse mit der Protokolldefinition aber zusätzliche Arbeit auf. Wenn Sie nicht sicher sind, ob eine Klasse eine bestimmte Methode implementiert oder nicht, müssen Sie dies erst überprüfen, bevor Sie eine Nachricht senden. Zum Glück geht das in Objective-C und der Klasse `NSObject` einfach:

```
// Optionale Clientmethode
if ([self.client respondsToSelector: @selector(nothingDidHappen)])
   [self.client nothingDidHappen];
```

`NSObject` stellt die Methode `respondsToSelector:` bereit, die den booleschen Wert `YES` zurückgibt, wenn ein Objekt die Methode implementiert, und anderenfalls `NO`. Wenn Sie den Client als `<NSObject>` deklarieren, teilen Sie dem Compiler mit, dass der Client diese Methode handhaben kann, sodass Sie den Client auf Kompatibilität prüfen können, bevor Sie ihm eine Methode senden.

3.10.6 Ein Protokoll erfüllen

Die Konformität mit einem Protokoll ist bei Klassen in ihrer Schnittstellendeklaration enthalten. Wenn ein Ansichtscontroller das Protokoll `JackClient` implementiert, so ist dies in spitzen Klammern angegeben. Eine Klasse kann auch mehreren Protokollen gehorchen. Dazu geben Sie alle Protokollnamen innerhalb des Klammernpaars an und trennen sie jeweils durch Kommata.

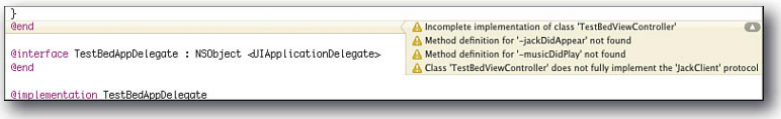
```
@interface TestBedViewController :
   UIViewController <JackClient>
{
   JackInTheBox *jack;
}
@property (retain) JackInTheBox *jack;
@end
```

Wenn Sie das Protokoll `JackClient` deklarieren, können Sie die Eigenschaft `client` des Hosts zuweisen. Der folgende Code wird fehlerlos kompiliert, da die Klasse für `self` im Einklang mit `JackClient` deklariert wurde.

```
self.jack = [JackInTheBox jack];
self.jack.client = self;
```

Hätten Sie die Protokolldeklaration in der Schnittstelle weggelassen, würde diese Zuweisung bei der Kompilierung zu einem Fehler führen.

Nachdem Sie das Protokoll in spitzen Klammern angegeben haben, *müssen* Sie dessen gesamten erforderlichen Methoden in Ihrer Klasse implementieren. Wenn Sie auch nur eine davon auslassen, führt das zu den Compiler-Warnungen, die Sie in *Abbildung 3.3* sehen. Der Compiler teilt Ihnen mit, welche Methoden fehlen und zu welchem Protokoll sie gehören.



▶ *Abbildung 3.3: Sie müssen alle erforderlichen Methoden implementieren, um ein Protokoll zu erfüllen. Objective-C gibt bei unvollständigen Implementierungen Warnungen aus.*

Die meisten Protokollmethoden im iPhone SDK sind optional. Sowohl die erforderlichen als auch die optionalen Methoden sind in der Entwicklerdokumentation ausführlich beschrieben. Beachten Sie, dass die Dokumentation der Protokolle von der Dokumentation der Klassen, die sie unterstützen, getrennt ist. Beispielsweise enthält die Xcode-Dokumentation drei verschiedene Referenzseiten für `UITableView`: eine für die Klasse `UITableView`, eine für das Protokoll `UITableViewDelegate` und eine für das Protokoll `UITableViewDataSource`.

3.11 Foundation-Klassen

Es gibt einige wenige Schlüsselklassen, mit denen Sie sich unbedingt vertraut machen müssen, bevor Sie sich auf das Programmieren stürzen, und das gilt vor allem, wenn Ihnen Objective-C noch neu ist. Dazu gehören Strings, Zahlen und Sammlungen, die entscheidende Bausteine für das Erstellen von Anwendungen bilden. Beispielsweise ist die Klasse `NSString` das Arbeitstier für fast alle Aufgaben der Textbearbeitung in Objective-C. Wie andere grundlegende Klassen ist sie jedoch nicht in Objective-C selbst definiert, sondern gehört zum Framework *Foundation*, in dem Sie nahezu alle wichtigen Klassen finden, die Sie für die tägliche Arbeit brauchen.

Foundation umfasst mehr als ein Dutzend Objektfamilien und Hunderte von Objektklassen – von Wertobjekten zum Speichern von Zahlen und Daten über Strings zum Festhalten von Zeichendaten und Sammlungen für andere Objekte bis zu Klassen, die auf das Dateisystem zugreifen und Daten von URLs abrufen. Foundation wird manchmal auch (nicht ganz korrekt) als Cocoa bezeichnet. (Cocoa und seine iPhone-Entsprechung Cocoa Touch enthalten tatsächlich alle Frameworks für die Mac OS X-Programmierung.) Wenn Sie Foundation beherrschen, beherrschen Sie auch die Programmierung in Objective-C, aber eine gründliche Abhandlung dieses Themas würde ein eigenes Buch ergeben.

Ich kann Ihnen in diesem Abschnitt keine umfassende Einführung in die Foundation-Klassen geben, sondern nur einen knappen Überblick als »Überlebenshilfe« für Programmierer. Im Folgenden finden Sie die Klassen, die Sie kennen müssen, und einige absolut notwendige Grundsätze, um mit ihnen zu arbeiten. Außerdem habe ich viele Codebeispiele beigefügt, die die Verwendung der einzelnen Klassen veranschaulichen, sodass Sie wenigstens einen Ausgangspunkt haben.

3.11.1 Strings

Wie ihre Vettern, die C-Strings vom Typ `(char *)`, dienen auch Cocoa-Strings zum Speichern von Zeichendaten. Allerdings handelt es sich bei ihnen um Objekte statt um Bytearrays. Anders als in C kann die Klasse `NSString` in Cocoa nicht verändert werden. Zwar können Sie Strings heranziehen, um daraus andere Strings zu erstellen, Sie können aber die Strings, die Sie bereits haben, nicht bearbeiten. Stringkonstanten sind durch Anführungszeichen und ein @-Symbol begrenzt. Im Folgenden sehen Sie eine typische Stringkonstante, die einer Stringvariable zugewiesen wird:

```
NSString *myString = @"A string constant";
```

Strings erstellen

Sie können Strings mit einer Formatierung erstellen, ganz ähnlich wie bei der Verwendung von `sprintf`. Wenn Sie bereits damit vertraut sind, `printf`-Anweisungen zu schreiben, können Sie Ihre Kenntnisse unmittelbar auf die Stringformatierung übertragen. Um Objekte in Strings aufzunehmen, verwenden Sie den Formatspezifizierer `%@`. Die verschiedenen Stringformatspezifizierer sind ausführlich im *Cocoa String Programming Guide* dokumentiert, den Sie über das Dokumentationsfenster von Xcode einsehen können ⌘ + Alt + ?. Die am häufigsten verwendeten Formate sind in *Tabelle 3.1* aufgeführt.

```
NSString *myString = [NSString stringWithFormat:
   @"The number is %d", 5];
```

Sie können Strings aneinanderhängen, um neue Strings zu bilden. Der folgende Aufruf gibt »The number is 522« aus. Dabei wird eine neue Instanz aus anderen Strings aufgebaut.

```
NSLog(@"%@", [myString stringByAppendingString:@"22"]);
```

Durch das Anhängen von Formaten gewinnen Sie noch weitere Möglichkeiten. Sie können den Formatstring und die Bestandteile angeben, aus denen das Ergebnis aufgebaut wird:

```
NSLog(@"%@", [myString stringByAppendingString:@"%d", 22]);
```

Längen und Zeichen mit Indexwert

Jeder String kann seine Länge angeben (über `length`) und auf Aufforderung ein Zeichen mit einem bestimmten Indexwert ausgeben. Die beiden folgenden Aufrufe führen zu der Ausgabe 15 bzw. e für den zuvor gezeigten String `@"The number is 5"`. Cocoa-Zeichen weisen den Typ `unichar` auf, in dem Unicode-Zeichen abgelegt werden.

```
NSLog(@"%d", myString.length);
printf("%c", [myString characterAtIndex:2]);
```

Konvertierung aus und in C-Strings

Auch wenn Sie in Objective-C arbeiten, machen sich häufig die Umstände der Programmierung in C bemerkbar, weshalb es sehr wichtig ist, zwischen C- und Cocoa-Strings wechseln zu können. Um einen `NSString` in einen C-String zu konvertieren, senden Sie ihm entweder `UTF8String` oder `cStringUsingEncoding:`. Beide Verfahren sind gleichwertig und ergeben dieselben C-Bytes:

```
printf("%s\n", [myString UTF8String]);
printf("%s\n", [myString cStringUsingEncoding: NSUTF8StringEncoding]);
```

Sie können auch umgekehrt vorgehen und einen C-String in einen `NSString` umwandeln, indem Sie `stringWithCString: encoding:` verwenden. Die hier gezeigten Beispiele weisen eine UTF-8-Kodierung auf, aber Objective-C unterstützt eine breite Palette von Möglichkeiten, z. B. ASCII, Japanisch, Lateinisch, Windows-CP1251 usw.

```
NSLog(@"%@", [NSString stringWithCString:"Hello World" encoding:
   NSUTF8StringEncoding]);
```

Strings in Dateien schreiben und aus Dateien lesen

Eine praktische Möglichkeit zum Speichern und Abrufen von Daten besteht darin, Strings in das lokale Dateisystem zu schreiben und von dort zu lesen. Das folgende Fragment zeigt, wie Sie einen String in eine Datei schreiben:

```
NSString *myString = @"Hello World";
NSError *error;
NSString *path = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/file.txt"];
if (![myString writeToFile:path atomically:YES
encoding:NSUTF8StringEncoding
error:&error])
{
   NSLog(@"Error writing to file: %@", [error localizedDescription]);
   return;
}
NSLog(@"File successfully written to file");
```

Der Pfad für die Datei ist `NSHomeDirectory()`, eine Funktion, die einen String mit einem Pfad zurückgibt, der auf die Anwendungs-Sandbox zeigt. Beachten Sie die besondere Methode, mit der der Teilpfad `Documents/file.txt` angehängt wird.

In Cocoa gibt es bei den meisten Routinen für den Dateizugriff eine Option, um den Vorgang als Einheit durchzuführen. Wenn Sie den Parameter `atomically` auf `YES` setzen, schreibt das iPhone die Datei in eine temporäre Hilfsdatei und benennt diese dann um. Durch einen solchen als Einheit durchgeführten Schreibvorgang vermeiden Sie eine Beschädigung der Datei.

Die hier gezeigte Anforderung gibt einen booleschen Wert zurück, nämlich `YES`, wenn der String geschrieben wurde, und `NO`, wenn dies nicht der Fall ist. Sollte die Schreibanforderung fehlschlagen, protokolliert dieser Code den Fehler mit einer übersetzten Beschreibung. Die Fehlerinformation wird in einer Instanz der Klasse `NSError` gespeichert, und der Selektor `localizedDescription` wird gesendet, um die Informationen in eine für den Menschen lesbare Form zu übersetzen. Wenn eine iPhone-Methode Fehler zurückgibt, können Sie mit diesem Ansatz bestimmen, welcher Fehler ausgelöst wurde.

Das Lesen eines Strings aus einer Datei folgt dem gleichen Muster, gibt aber kein boolesches Ergebnis zurück. Stattdessen müssen Sie überprüfen, ob der zurückgegebene String `nil` ist, und den zurückgegebenen Fehler anzeigen, wenn dies der Fall ist.

```
NSString *inString = [NSString stringWithContentsOfFile:path
encoding:NSUTF8StringEncoding error:&error];
if (!inString)
{
NSLog(@"Error reading from file % %@", [path lastPathComponent],
[error localizedDescription]);
return;
}
NSLog(@"File successfully read from file");
NSLog(@"%@", inString);
```

Zugriff auf Teilstrings

In Cocoa gibt es eine Reihe von Möglichkeiten, um Teilstrings aus Strings zu extrahieren. Die folgende kurze Übersicht zeigt einige der typischen Ansätze. Wie Sie wahrscheinlich erwarten, nimmt die Bearbeitung von Strings einen großen Teil jeder flexiblen API ein. Cocoa bietet sehr viel mehr Routinen und Klassen, um Strings zu analysieren und zu interpretieren, als die wenigen, die hier aufgeführt sind. In dieser groben Übersicht über `NSString` werden z. B. `NSScanner`, `NSXMLParser` usw. nicht erwähnt.

Strings in Arrays konvertieren

Sie können einen String in ein Array umwandeln, indem Sie seine Bestandteile an dem dazwischen auftretenden Begrenzungszeichen abtrennen. Im folgenden Beispiel wird der String an den Leerzeichen aufgetrennt und in einzelne Wörter zerlegt. Die Leerzeichen werden verworfen, sodass nur ein Array mit den einzelnen Zahlwörtern übrig bleibt.

```
NSString *myString = @"One Two Three Four Five Six Seven";
NSArray *wordArray = [myString componentsSeparatedByString: @" "];
NSLog(@"%@", wordArray);
```

Teilstrings nach Index abrufen

Sie können Teilstrings vom Anfang eines Strings bis zu einem bestimmten Index oder von einem Index bis zum Ende anfordern. Bei den beiden folgenden Beispielen mit den `to`- und `from`-Versionen der Teilstringindexabfrage wird `@"One Two"` bzw. `@"Two Three Four Five Six Seven"` zurückgegeben. Wie in Standard-C beginnen die Indizes von Arrays und Strings bei 0.

```
NSString *sub1 = [myString substringToIndex:7];
NSLog(@"%@", sub1);
NSString *sub2 = [myString substringFromIndex:4];
NSLog(@"%@", sub2);
```

Teilstrings aus Bereichen gewinnen

Mithilfe von Bereichen können Sie genau angeben, wo ein Teilstring anfangen und aufhören soll. Der folgende Code gibt `@"Tw"` zurück, was beim Zeichen mit dem Index 4 beginnt und zwei Zeichen lang ist. In `NSRange` können Sie einen Abschnitt innerhalb einer Zeichenfolge definieren. Solche Bereiche setzen Sie bei indizierenden Elementen wie Strings und Arrays ein.

```
NSRange r;
r.location = 4;
r.length = 2;
NSString *sub3 = [myString substringWithRange:r];
NSLog(@"%@", sub3);
```

Suchen und ersetzen in Strings

In Cocoa können Sie einen String auf einfache Weise nach einem Teilstring durchsuchen. Die Suche gibt einen Bereich zurück, der durch eine Anfangsposition und eine Länge gekennzeichnet ist. Überprüfen Sie stets die Anfangsposition! Die Position `NSNotFound` bedeutet, dass die Suche fehlgeschlagen ist. Der folgende Code gibt eine Position von 18 und eine Länge von 4 zurück:

```
NSRange searchRange = [myString rangeOfString:@"Five"];
if (searchRange.location != NSNotFound)
   NSLog(@"Range location: %d, length: %d", searchRange.location,
      searchRange.length);
```

Nachdem Sie einen Bereich gefunden haben, können Sie ihn durch einen neuen String ersetzen. Dabei muss der neue String nicht genauso lang sein wie der ursprüngliche, sodass der resultierende String länger oder kürzer sein kann als derjenige, von dem Sie ausgegangen sind.

```
NSLog(@"%@", [myString stringByReplacingCharactersInRange:
   searchRange withString: @"New String"]);
```

Mit einem allgemeineren Ansatz können Sie alle Vorkommen eines bestimmten Strings ersetzen. Das folgende Codefragment tauscht die einzelnen Leerzeichen durch ein Muster mit einem Sternchen aus, sodass sich `@"One * Two * Three * Four * Five * Six * Seven"` ergibt:

```
NSString *replaced = [myString stringByReplacingOccurrencesOfString:
@" " withString: @" * "];
NSLog(@"%@", replaced);
```

Groß- und Kleinschreibung ändern

Cocoa bietet drei einfache Methoden, um die Groß- und Kleinschreibung eines Strings zu ändern. Die drei folgenden Beispiele geben den String einmal vollständig in Großbuchstaben, einmal vollständig in Kleinbuchstaben sowie in der Form englischer Überschriften zurück, bei denen jedes Wort mit einem Großbuchstaben beginnt (»Hello World. How Do You Do?«). Da Cocoa Vergleiche ohne Berücksichtigung der Groß- und Kleinschreibung ermöglicht, werden Sie selten zwischen Groß- und Kleinschreibung umwandeln müssen, wenn Sie Strings miteinander vergleichen.

```
NSString *myString = @"Hello world. How do you do?";
NSLog(@"%@", [myString uppercaseString]);
NSLog(@"%@", [myString lowercaseString]);
NSLog(@"%@", [myString capitalizedString]);
```

Strings prüfen

Auf dem iPhone gibt es verschiedene Möglichkeiten, um Strings zu vergleichen und zu prüfen. Die drei einfachsten sind die Überprüfung auf Gleichheit zweier Strings, auf Übereinstimmung mit einem Stringpräfix (den Zeichen, mit denen der String beginnt) bzw. dem Suffix (den Zeichen, mit denen der String endet). Bei komplizierteren Vergleichen werden `NSComparisonResults`-Konstanten verwendet, um anzuzeigen, wie die einzelnen Elemente im Vergleich zueinander angeordnet sind.

```
NSString *s1 = @"Hello World";
NSString *s2 = @"Hello Mom";
NSLog(@"%@ %@ %@", s1, [s1 isEqualToString:s2] ?
   @"equals" : @"differs from", s2);
NSLog(@"%@ %@ %@", s1, [s1 hasPrefix:@"Hello"] ?
   @"starts with" : @"does not start with", @"Hello");
NSLog(@"%@ %@ %@", s1, [s1 hasSuffix:@"Hello"] ?
   @"ends with" : @"does not end with", @"Hello");
```

Zahlen aus Strings gewinnen

Sie können Strings mithilfe einer Wertmethode in Zahlen umwandeln. Die folgenden Beispiele geben 3, 1, 3,141592 und 3,141592 zurück:

```
NSString *s1 = @"3.141592";
NSLog(@"%d", [s1 intValue]);
NSLog(@"%d", [s1 boolValue]);
NSLog(@"%f", [s1 floatValue]);
NSLog(@"%f", [s1 doubleValue]);
```

Veränderbare Strings

`NSMutableString` ist eine Unterklasse von `NSString` und bietet Ihnen die Möglichkeit, mit Strings zu arbeiten, deren Inhalte geändert werden können. Nachdem ein String instanziiert ist, können Sie neue Inhalte an ihn anhängen, um so ein Ergebnis zusammenzubauen, bevor es von einer Methode zurückgegeben wird. Das folgende Beispiel zeigt »Hello World. The results are in now.« an.

```
NSMutableString *myString = [NSMutableString stringWithString:
   @"Hello World. "];
[myString appendFormat:@"The results are %@ now.", @"in"];
NSLog(@"%@", myString);
```

3.11.2 Zahlen und Datumsangaben

Foundation umfasst auch eine große Familie von Wertklassen, darunter auch solche für Zahlen und Datumsangaben. Anders als die Fließkommazahlen, Integer usw. von Standard-C sind all dies Objekte. Sie können zugewiesen und freigegeben und in Sammlungen wie Arrays, Dictionarys und Mengen verwendet werden. Die folgenden Beispiele zeigen Zahlen und Datumsangaben in Aktion und geben einen grundlegenden Überblick über diese Klassen.

Mit Zahlen arbeiten

Mit der Klasse `NSNumber` können Sie Zahlen als Objekte behandeln. Neue Instanzen von `NSNumber` erstellen Sie mit einer Reihe von Hilfsmethoden, z. B. mit `numberWithInt:`, `numberWithFloat:`, `numberWithBool:` usw. Wenn die Werte einmal gesetzt sind, können Sie sie mit `intValue`, `floatValue`, `boolValue` usw. abrufen und mit normalen mathematischen Operationen von C Berechnungen damit anstellen.

Sie sind nicht darauf beschränkt, ein Objekt mit demselben Datentyp abzurufen, mit dem es gesetzt wurde, sondern können z. B. einen Fließkommawert festlegen und daraus einen Integer abrufen. Zahlen lassen sich auch in Strings konvertieren.

```
NSNumber *number = [NSNumber numberWithFloat:3.141592];
NSLog(@"%d", [number intValue]);
NSLog(@"%@", [number stringValue]);
```

Einer der wichtigsten Gründe für die Verwendung von `NSNumber`-Objekten anstelle von `int`- und `float`-Werten usw. besteht darin, dass Sie sie in Cocoa-Routinen und -Klassen verwenden können. Beispielweise können Sie keinen Benutzerstandardwert (also eine Voreinstellung) auf den Integerwert 23 setzen, etwa für die Meldung: »Sie haben dieses Programm bereits 23-mal verwendet.« Es ist jedoch möglich, das Objekt `[NSNumber numberWithInt:23]` zu speichern und später den Integerwert aus diesem Objekt abzurufen, um daraus eine Meldung für den Benutzer zu machen.

> **Hinweis**
>
> Die Klasse `NSDecimalNumber` stellt einen praktischen objektorientierten Wrapper für die Arithmetik von Dezimalzahlen dar.

Arbeiten mit Datumsangaben

Wie im standardmäßigen C und bei `time()` wird auch in `NSDate`-Objekten die Anzahl der Sekunden seit einer Epoche, also einer genormten globalen Zeitreferenz, zur Darstellung des aktuellen Datums verwendet. Die iPhone-Epoche begann um Mitternacht am 1. Januar 2001, die Unix-Standardepoche begann um Mitternacht am 1. Januar 1970.

Jedes `NSTimeInterval` stellt eine Zeitspanne in Sekunden dar, die mit einer Fließkommagenauigkeit im Bereich von Sekundenbruchteilen gespeichert werden. Der folgende Code zeigt, wie Sie ein neues Datumsobjekt mit der aktuellen Zeit erstellen und wie Sie mithilfe eines Intervalls auf einen Zeitpunkt in der Zukunft (oder Vergangenheit) verweisen:

```
// Aktueller Zeitpunkt
NSDate *date = [NSDate date];

// Zeitpunkt zehn Sekunden in der Zukunft
date = [NSDate dateWithTimeIntervalSinceNow:10.0f];
```

Datumsangaben vergleichen Sie, indem Sie das Zeitintervall dazwischen festlegen bzw. überprüfen. Das folgende Codefragment zwingt die Anwendung dazu, fünf Sekunden lang zu ruhen, und vergleicht dann das aktuelle Datum mit dem in `date` gespeicherten:

```
// Ruht fünf Sekunden lang und prüft dann das Zeitintervall
[NSThread sleepUntilDate:[NSDate dateWithTimeIntervalSinceNow:5.0f]];
NSLog(@"Slept %f seconds", [[NSDate date] timeIntervalSinceDate:date]);
```

Die standardmäßige Beschreibungsmethode für Datumsangaben gibt einen für Menschen lesbaren String zurück, der das aktuelle Datum und die Uhrzeit zeigt:

```
// Zeigt das Datum
NSLog(@"%@", [date description]);
```

Um Datumsangaben in rundum formatierte Strings zu verwandeln, anstatt einfach die Standardbeschreibung zu verwenden, setzen Sie eine Instanz von `NSDateFormatter` ein. Sie geben das Format (z. B. `YY` für die Jahresangabe mit zwei Stellen und `YYYY` für die mit vier Stellen) über die Eigenschaft `dateFormat` des Objekts an. Eine vollständige Liste der Formatspezifizierer finden Sie in der mitgelieferten Xcode-Dokumentation. Mit dieser Klasse können Sie nicht nur für eine formatierte Ausgabe sorgen, sondern auch vorformatierte Datumsausgaben aus Strings lesen. Dies überlassen wir Ihnen als Übungsaufgabe.

```
// Gibt einen formatierten String aus, der für das aktuelle Datum steht
NSDateFormatter *formatter = [[[NSDateFormatter alloc] init]
   autorelease];
formatter.dateFormat = @"MM/dd/YY HH:mm:ss";
NSString *timestamp = [formatter stringFromDate:[NSDate date]];
NSLog(@"%@", timestamp);
```

Timer

Manchmal ist es notwendig, dafür zu sorgen, dass eine Aktion zu einem bestimmten Zeitpunkt in der Zukunft ausgeführt wird. Cocoa bietet einen einfach zu verwendenden Timer, der nach dem von Ihnen angegebenen Zeitintervall ausgelöst wird, nämlich die Klasse `NSTimer`. Der im Folgenden gezeigte Timer wird nach einer Sekunde wiederholt ausgelöst, bis er deaktiviert wird:

```
[NSTimer scheduledTimerWithTimeInterval: 1.0f target: self selector:
@selector(handleTimer:) userInfo: nil repeats: YES];
```

Jedes Mal, wenn der Timer ausgelöst wird, ruft er sein Ziel auf und sendet ihm die Selektornachricht, mit der er initialisiert wurde. Die Callback-Methode nimmt ein Argument entgegen (beachten Sie den einzelnen Doppelpunkt), nämlich den Timer selbst. Um einen Timer zu deaktivieren, senden Sie ihm die Nachricht `invalidate`. Dadurch wird das Timerobjekt freigegeben und aus der aktuellen Ausführungsschleife entfernt.

```
- (void) handleTimer: (NSTimer *) timer
{
   printf("Timer count: %d\n", count++);
   if (count > 3)
   {
      [timer invalidate];
      printf("Timer disabled\n");
   }
}
```

Informationen über Indexpfade abrufen

Die Klasse `NSIndexPath` wird bei iPhone-Tabellen verwendet. In ihr sind die Abschnitts- und die Zeilennummer einer Benutzerauswahl gespeichert, also die Stelle, auf die ein Benutzer auf der Tabelle tippt. Wenn diese Zahlen mit Indexpfaden versehen sind, können Sie sie mit den Eigenschaften `myIndexPath.row` und `myIndexPath.section` abrufen. Weitere Informationen über diese Klasse und ihre Verwendung erhalten Sie in Kapitel 11, *Tabellenansichten erstellen und verwalten*.

3.11.3 Sammlungen

Auf dem iPhone werden hauptsächlich drei Arten von Sammlungen verwendet: Arrays, Dictionarys und Mengen (Sets). Arrays verhalten sich wie C-Arrays. Sie bestehen aus einer indizierten Liste von Objekten, die Sie abrufen können, indem Sie angeben, bei welchem Index nachgeschlagen werden soll. In Dictionarys sind Werte dagegen so gespeichert, dass Sie sie anhand von Schlüsseln nachschlagen können. Beispielsweise können Sie das Alter von Personen in einem Dictionary ablegen, wobei das Alter des Vaters das `NSNumber`-Objekt 57 und das Alter des Kindes das Objekt 15 ist. Mengen sind ungeordnete Gruppen von Objekten und werden auf dem iPhone vor allem in Verbindung mit dem Lesen von Benutzerberührungen vom Bildschirm eingesetzt. Von jeder dieser Klassen gibt es wie bei `NSString` eine normale und eine veränderbare Version.

Arrays erstellen und darauf zugreifen

Arrays erstellen Sie mit der Hilfsmethode `arrayWithObjects:`, die ein zur automatischen Freigabe gekennzeichnetes Array zurückgibt. Wenn Sie diese Methode aufrufen, führen Sie alle Objekte auf, die Sie zu dem Array hinzufügen möchten, und schließen die Liste mit `nil` ab. (Wenn Sie auf das `nil` verzichten, stürzt die Anwendung zur Laufzeit ab.) Einem Array können Sie jede beliebige Art von Objekt hinzufügen, auch andere Arrays und Dictionarys. Das folgende Beispiel zeigt, wie Sie ein Array mit drei Elementen erstellen:

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
```

Die Eigenschaft `count` gibt die Anzahl der Objekte im Array an. Arrays werden beginnend bei 0 indiziert, sodass der höchste Indexwert um 1 kleiner ist als der Wert von `count`. Wenn Sie versuchen, auf `[array objectAtIndex:array.count]` zuzugreifen, führt dies zu einer Ausnahme aufgrund einer Indexüberschreitung und zum Absturz. Seien Sie beim Abrufen von Objekten also stets vorsichtig, und achten Sie darauf, weder die obere noch die untere Grenze für das Array zu überschreiten.

```
NSLog(@"%d", array.count);
NSLog(@"%@", [array objectAtIndex:0]);
```

Die veränderbare Variante von `NSArray` heißt `NSMutableArray`. Veränderbare Arrays lassen sich bearbeiten, sodass Sie nach Belieben Objekte hinzufügen oder daraus entfernen können. Der folgende Code kopiert das vorstehende Array in ein veränderbares und bearbeitet Letzteres, indem er ein Objekt hinzufügt und ein anderes entfernt. Dadurch ergibt sich das Array `[@"One", @"Two", @"Four"]`:

```
NSMutableArray *marray = [NSMutableArray arrayWithArray:array];
[marray addObject:@"Four"];
[marray removeObjectAtIndex:2];
NSLog(@"%@", marray);
```

Sowohl veränderbare als auch nicht veränderbare Arrays können Sie stets zu einem neuen kombinieren, das alle Elemente der einzelnen Arrays enthält. Hierbei wird nicht geprüft, ob dabei Duplikate auftreten. Der folgende Code erstellt ein Array aus sechs Elementen, das die Zahlwörter *one*, *two* und *three* des ursprünglichen Arrays und *one*, *two* und *four* des veränderbaren Arrays enthält:

```
NSLog(@"%@", [array arrayByAddingObjectsFromArray:marray]);
```

Arrays prüfen

Sie können prüfen, ob ein Array ein bestimmtes Objekt enthält, und den Index eines gegebenen Objekts abrufen. Der folgende Code sucht nach dem ersten Vorkommen von »Four« und gibt den Index des betreffenden Objekts zurück. Der Test in der `if`-Anweisung stellt sicher, dass es mindestens eine solche Fundstelle gibt.

```
if ([marray containsObject:@"Four"])
   NSLog(@"The index is %d",
      [marray indexOfObject:@"Four"]);
```

Arrays in Strings umwandeln

Wie zu anderen Objekten können Sie auch zu einem Array die Nachricht `description` senden, woraufhin ein `NSString` mit einer Beschreibung des Arrays zurückgegeben wird. Außerdem können Sie ein `NSArray` mit `componentsJoinedByString` in einen String umwandeln. Der folgende Code gibt `@"One Two Three"` zurück.

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
NSLog(@"%@", [array componentsJoinedByString:@" "]);
```

Dictionarys erstellen und darauf zugreifen

In `NSDictionary`-Objekten werden Schlüssel und Werte gespeichert, sodass Sie die Objekte mithilfe von Strings nachschlagen können. Die veränderbare Version von Dictionarys, `NSMutableDictionary`, erlaubt es Ihnen, diese Dictionarys zu bearbeiten, indem Sie bei Bedarf Elemente hinzufügen und entfernen. Bei der iPhone-Programmierung verwenden Sie die veränderbare Klasse häufiger als die statische, weshalb in diesen Beispielen die veränderbaren Versionen vorgeführt werden.

Dictionarys erstellen

Mit der Hilfsmethode `dictionary` erstellen Sie ein neues veränderbares Dictionary, wie Sie im Folgenden sehen. Dadurch wird ein neu initialisiertes Dictionary zurückgegeben, das Sie bearbeiten können. Sie füllen es mit `setObject: forKey:`.

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
[dict setObject:@"1" forKey:@"A"];
[dict setObject:@"2" forKey:@"B"];
[dict setObject:@"3" forKey:@"C"];
NSLog(@"%@", [dict description]);
```

Dictionarys durchsuchen

Ein Dictionary zu durchsuchen bedeutet, es nach dem Schlüsselnamen abzufragen. Sie verwenden `objectForKey:`, um das Objekt zu finden, das zu dem gegebenen Schlüssel gehört. Wird der Schlüssel nicht gefunden, gibt das Dictionary `nil` zurück.

```
NSLog(@"%@", [dict objectForKey:@"A"]);
NSLog(@"%@", [dict objectForKey:@"F"]);
```

Objekte ersetzen

Wenn Sie für einen bereits belegten Schlüssel ein neues Objekt festlegen, ersetzt Cocoa das ursprüngliche Objekt im Dictionary. Der folgende Code ersetzt den Wert `3` für den Schlüssel `C` durch `foo`:

```
[dict setObject:@"foo" forKey:@"C"];
NSLog(@"%@", [dict objectForKey:@"C"]);
```

Objekte entfernen

Sie können Objekte auch aus Dictionarys entfernen. Mit dem folgenden Code löschen Sie das Objekt, das zum Schlüssel `B` gehört. Anschließend sind sowohl der Schlüssel als auch das Objekt nicht mehr im Dictionary vorhanden.

```
[dict removeObjectForKey:@"B"];
```

Schlüssel auflisten

Dictionarys können die Anzahl der in ihnen gespeicherten Einträge melden und ein Array aller zurzeit verwendeten Schlüssel ausgeben. Anhand dieser Schlüsselliste können Sie ablesen, welche Schlüssel schon in Gebrauch sind. Bevor Sie ein Element zum Dictionary hinzufügen, können Sie einen Vergleich mit dieser Liste durchführen, damit Sie kein vorhandenes Schlüssel/Wert-Paar überschreiben.

```
NSLog(@"The dictionary has %d objects", [dict count]);
NSLog(@"%@", [dict allKeys]);
```

Zugriff auf Mengenobjekte

In Mengen (Sets) sind ungeordnete Objektsammlungen gespeichert. Mengen begegnen Ihnen fast ausschließlich bei der Arbeit mit dem Multitouch-Bildschirm des iPhones. Die Klasse `UIView` empfängt Aktualisierungen von Fingerbewegungen, die Berührungen als `NSSet` ausgeben. Um solche Berührungen zu handhaben, müssen Sie fast immer `allObjects` verwenden und mit dem daraufhin zurückgegebenen Array arbeiten. Nach dieser Konvertierung können Sie normale Arrayaufrufe verwenden, um die Berührungen aufzulisten, abzufragen und zu durchlaufen.

Speicherverwaltung bei Sammlungen

Arrays, Mengen und Dictionarys behalten Objekte, die zu ihnen hinzugefügt werden, automatisch bei, und geben solche frei, die aus ihnen entfernt werden. Freigabenachrichten werden auch gesendet, wenn die Zuweisung der Sammlung aufgehoben wird. Sammlungen kopieren keine Objekte, sondern verlassen sich auf die Beibehaltungszähler, um die Objekte festzuhalten und nach Bedarf zu verwenden.

Sammlungen in eine Datei schreiben

Sowohl Arrays als auch Dictionarys können mit der Methode `writeToFile: atomically:` in Dateien gespeichert werden, sofern die Typen innerhalb der Sammlung `NSData`, `NSDate`, `NSNumber`, `NSString`, `NSArray` oder `NSDictionary` sind. Als erstes Argument übergeben Sie den Pfad, als zweites einen booleschen Wert. Wie beim Speichern von Strings bestimmt das zweite Argument, ob die Datei zuerst als temporäre Hilfsdatei gespeichert und dann umbenannt werden soll. Die Methode gibt den booleschen Wert `YES` zurück, wenn die Datei gespeichert wurde, und anderenfalls `NO`. Zum Speichern von Arrays und Dictionarys werden standardmäßige Eigenschaftslistendateien erstellt.

```
NSString *path = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/ArraySample.txt"];
if ([array writeToFile:path atomically:YES])
   NSLog(@"File was written successfully");
```

Um ein Array oder ein Dictionary aus einer Datei abzurufen, verwenden Sie die Hilfsmethode `arrayWithContentsOfFile:` bzw. `dictionaryWithContentsOfFile:`. Gibt die Methode `nil` zurück, konnte die Datei nicht gelesen werden.

```
NSArray *newArray = [NSArray arrayWithContentsOfFile:path];
NSLog(@"%@", newArray);
```

URLs konstruieren

`NSURL`-Objekte zeigen auf Ressourcen, bei denen es sich um lokale Dateien, aber auch um URLs im Web handeln kann. Um URL-Objekte zu erstellen, übergeben Sie einen String an eine Klassen-Hilfsfunktion, wobei es für die einzelnen Arten von URLs unterschiedliche Funktionen gibt. Ist ein `NSURL`-Objekt aber einmal erstellt, kann es nicht mehr verändert werden. Cocoa kümmert sich nicht darum, ob die Ressource lokal ist oder auf ein Objekt zeigt, das nur im Internet existiert. Der folgende Code zeigt, wie Sie URLs für die beiden Typen – lokaler Pfad und Web – erstellen:

```
NSString *path = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/foo.txt"];
NSURL *url1 = [NSURL fileURLWithPath:path];
NSLog(@"%@", url1);

NSString *urlpath = @"http://ericasadun.com";
NSURL *url2 = [NSURL URLWithString:urlpath];
NSLog(@"%d characters read",
   [[NSString stringWithContentsOfURL:url2] length]);
```

Mit NSData arbeiten

Wenn `NSString`-Objekte die Entsprechung der mit o abgeschlossenen C-Strings sind, dann sind `NSData` mit Puffern zu vergleichen. `NSData` bietet Datenobjekte zum Speichern und Handhaben von Bytes. Häufig müssen Sie `NSData` mit dem Inhalt einer Datei eines URLs füllen. Die zurückgegebenen Daten können die Länge angeben, damit Sie wissen, wie viele Bytes abgerufen wurden. Das folgende Codefragment ruft den Inhalt eines URLs ab und gibt die Anzahl der gelesenen Bytes aus:

```
NSData *data = [NSData dataWithContentsOfURL:url2];
NSLog(@"%d", [data length]);
```

Um auf den grundlegenden Bytepuffer eines `NSData`-Objekts zuzugreifen, verwenden Sie `bytes`. Dadurch wird ein `(const void *)`-Zeiger auf die eigentlichen Daten zurückgegeben.

Wie bei anderen Cocoa-Objekten können Sie einerseits die standardmäßige Version der Klasse `NSData` verwenden, andererseits aber auch die veränderbare Kindklasse `NSMutableData`. Die meisten Cocoa-Programme, die auf das Web zugreifen, vor allem diejenigen, die asynchrone Downloads durchführen, »ziehen« immer nur wenige Daten auf einmal. Für diese Fälle sind `NSMutableData`-Objekte sinnvoll. Sie können veränderbare Daten langsam anwachsen lassen, indem Sie `appendData:` verwenden, um neue Informationen anzuhängen, sobald sie empfangen worden sind.

Dateiverwaltung

Der Dateimanager für das iPhone ist ein Singleton, das von der Klasse `NSFileManager` bereitgestellt wird. Es kann den Inhalt von Ordnern auflisten, um zu bestimmen, welche Dateien vorhanden sind, und grundlegende Aufgaben des Dateisystems durchführen. Der folgende Code ruft eine Dateiliste für zwei Ordner ab. Zuerst schaut er in den `Documents`-Ordner der Sandbox und dann in das Anwendungsbundle selbst.

```
NSFileManager *fm = [NSFileManager defaultManager];

// Führt die Dateien im Documents-Ordner der Sandbox auf
NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"
   Documents"];
NSLog(@"%@",[fm directoryContentsAtPath:path]);


// Führt die Dateien im Anwendungsbundle auf
path = [[NSBundle mainBundle] bundlePath];
NSLog(@"%@",[fm directoryContentsAtPath:path]);
```

Beachten Sie hier die Verwendung von `NSBundle`. Dadurch können Sie das Anwendungsbundle finden und dessen Pfad dem Dateimanager übergeben. Außerdem können Sie `NSBundle` verwenden, um den Pfad für ein Element abzurufen, das in dem Bundle enthalten ist. (Sie können jedoch niemals in das Anwendungsbundle schreiben.) Der folgende Code gibt den Pfad zum Bild `Default.png` der Anwendung zurück. Beachten Sie, dass der Dateiname und die Erweiterung jeweils für sich stehen und dass bei beiden auf die Groß- und Kleinschreibung geachtet wird.

```
NSBundle *mb = [NSBundle mainBundle];
NSLog(@"%@", [mb pathForResource:@"Default" ofType:@"png"]);
```

Der Dateimanager bietet die ganze Palette dateispezifischer Aufgaben. Er kann Dateien verschieben, kopieren und entfernen sowie das System nach Merkmalen und dem Besitzer von Dateien abfragen. Die folgenden Beispiele zeigen einige der einfacheren Routinen, die Sie in Ihren Anwendungen einsetzen können:

```
// Eine Datei erstellen
NSString *docspath = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents"];
NSString *filepath = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/testfile"];
```

```
NSArray *array = [@"One Two Three" componentsSeparatedByString:@" "];
[array writeToFile:filepath atomically:YES];
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);


// Eine Datei kopieren
NSString *copypath = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/copied"];
if (![fm copyItemAtPath:filepath toPath:copypath error:&error])
{
   NSLog(@"Copy Error: %@", [error localizedDescription]);
   return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);


// Eine Datei verschieben
NSString *newpath = [NSHomeDirectory()
stringByAppendingPathComponent:@"Documents/renamed"];
if (![fm moveItemAtPath:filepath toPath:newpath error:&error])
{
   NSLog(@"Move Error: %@", [error localizedDescription]);
   return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);


// Eine Datei entfernen
if (![fm removeItemAtPath:copypath error:&error])
{
   NSLog(@"Remove Error: %@", [error localizedDescription]);
   return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);
```

Hinweis

Ein weiterer bequemer Kniff für den Umgang mit Dateien besteht darin, in Pfadnamen eine Tilde zu verwenden, z. B. `"~/Library/Preferences/foo.plist"`, und die `NSString`-Methode `stringByExpandingTildeInPath` anzuwenden.

3.12 Zum guten Schluss: Nachrichtenweiterleitung

Objective-C bietet zwar keine echte mehrfache Vererbung, aber eine Notlösung, damit Objekte auf Nachrichten reagieren können, die in anderen Klassen implementiert sind. Wenn ein Objekt auf die Nachrichten einer anderen Klasse reagieren können soll, können Sie in Ihrer Anwendung die Nachrichtenweiterleitung verwenden, um Zugriff auf die Methoden des Objekts zu erhalten.

Normalerweise führt es zu einem Laufzeitfehler, wenn Sie eine nicht erkannte Nachricht senden, sodass die Anwendung abstürzt. Doch vor dem Absturz gibt das Laufzeitsystem des iPhones jedem Objekt eine zweite Chance, um die Nachricht zu handhaben. Wenn Sie die Nachricht abfangen, können Sie sie zu dem Objekt umleiten, das sie versteht und darauf reagieren kann.

Kehren wir zu der Beispielklasse `Car` zurück, die wir in diesem Kapitel behandelt haben. Die auf halbem Wege eingeführte Eigenschaft `carInfo` gibt einen String zurück, der den Hersteller, das Modell und das Baujahr des Autos beschreibt. Nehmen wir nun an, dass eine Instanz von `Car` auf `NSString`-Nachrichten antworten kann, indem sie sie an diese Eigenschaft weiterleitet. Wenn Sie dann `length` an ein `Car`-Objekt senden, stürzt die Anwendung nicht ab – stattdessen gibt das Objekt die Länge des `carInfo`-Strings zurück. Senden Sie `stringByAppendingString:`, so hängt das Objekt den String an den Eigenschaftsstring an. Dies wäre so, als hätte die Klasse `Car` das gesamte Stringverhalten geerbt (oder zumindest geliehen).

Objective-C ermöglicht ein solches Verhalten durch die sogenannte *Nachrichtenweiterleitung*. Wenn Sie eine Nachricht an ein Objekt senden, das mit dem Selektor nicht umgehen kann, wird dieser Selektor an die Methode `forwardInvocation:` weitergeleitet. Das mit dieser Nachricht gesendete Objekt, eine `NSInvocation`-Instanz, speichert den ursprünglichen Selektor und die anforderten Argumente. Sie können `forwardInvocation:` überschreiben und die Nachricht an ein anderes Objekt senden.

3.12.1 Die Nachrichtenweiterleitung umsetzen

Um die Nachrichtenweiterleitung in einem eigenen Programm umzusetzen, müssen Sie zwei Methoden überschreiben, nämlich `methodSignatureForSelector:` und `forwardInvocation:`. Die erste dieser beiden Methoden erstellt eine gültige Methodensignatur für Nachrichten, die von einer anderen Klasse implementiert werden, die zweite leitet den Selektor an ein Objekt weiter, das die Nachricht tatsächlich implementiert.

Eine Methodensignatur erstellen

Die erste Methode gibt eine Methodensignatur für den angeforderten Selektor zurück. In unserem Beispiel kann eine Instanz von `Car` keine korrekte Signatur für einen Selektor erstellen, der von einer anderen Klasse implementiert wird, in diesem Fall von `NSString`. Wenn Sie eine Überprüfung auf eine nicht wohlgeformte Signatur hinzufügen (also auf die Rückgabe von `nil`), geben Sie der

Methode die Gelegenheit, die einzelnen Möglichkeiten der Pseudovererbung durchzugehen und zu versuchen, ein gültiges Ergebnis hervorzurufen. Im folgenden Beispiel werden Methoden nur einer einzigen anderen Klasse über `self.carInfo` herangezogen:

```
- (NSMethodSignature*) methodSignatureForSelector:(SEL)selector
{
   // Prüft nach, ob Car die Nachricht handhaben kann
   NSMethodSignature* signature = [super
      methodSignatureForSelector:selector];
   // Wenn nicht, wird geprüft, ob der String carInfo etwas mit der
   // Nachricht anfangen kann
   if (!signature)
      signature = [self.carInfo methodSignatureForSelector:selector];

   return signature;
}
```

Weiterleiten

Die zweite Methode, die Sie überschreiben müssen, ist `forwardInvocation`. Sie wird nur dann aufgerufen, wenn ein Objekt nicht in der Lage war, mit einer Nachricht umzugehen. Diese Methode gibt dem Objekt eine zweite Chance, indem sie ihm erlaubt, die Nachricht weiterzuleiten. Sie prüft, ob der String `self.carInfo` auf den Selektor reagiert. Wenn ja, weist sie das aufrufende Objekt an, sich selbst mit dem Stringobjekt als Empfänger aufzurufen.

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
   SEL selector = [invocation selector];

   if ([self.carInfo respondsToSelector:selector])
   {
      printf("[forwarding from %s to %s] ", [[[self class] description]
         UTF8String], [[NSString description] UTF8String]);
      [invocation invokeWithTarget:self.carInfo];
   }
}
```

Weitergeleitete Nachrichten verwenden

Der Aufruf von Nicht-Klassenmethoden wie `UTF8String` oder `length` führt zu Compiler-Warnungen, die Sie aber ignorieren können. Der in *Abbildung 3.4* gezeigte Code ruft zwei solche Warnungen hervor, wird aber kompiliert und (was schließlich wichtiger ist) fehlerfrei ausgeführt. Wie Sie in der Abbildung sehen, können Sie einer `Car`-Instanz sowohl Methoden senden, die von der Klasse selbst definiert sind, als auch solche, die `NSString` implementiert.

▶ *Abbildung 3.4: Der Compiler gibt Warnungen über weitergeleitete Methoden aus, der Code läuft aber fehlerfrei.*

Aufräumen

Die Aufrufweiterleitung ahmt zwar die mehrfache Vererbung nach, doch `NSObject` setzt diese beiden Verfahren nicht gleich. Methoden wie `respondsToSelector:` und `isKindOfClass:` überprüfen nur die Vererbungshierarchie und kümmern sich nicht um die Weiterleitung.

Es gibt einige optionale Methoden, mit denen eine Klasse ihre Nachrichtenkonformität mit einer anderen Klasse besser ausdrücken kann. Wenn Sie `respondsToSelector:` und `isKindOfClass:` neu implementieren, können andere Klassen Ihre Klasse abfragen. Im Gegenzug gibt die Klasse bekannt, dass sie nicht nur auf ihre eigenen Methoden, sondern auch auf alle Stringmethoden reagiert und »eine Art von« String (»kind of«) ist. Dadurch wird der Ansatz der Pseudovererbung noch erweitert.

```
// Erweitert Selektorkonformität
- (BOOL)respondsToSelector:(SEL)aSelector
{
   // Die Klasse Car kann die Nachricht handhaben
   if ( [super respondsToSelector:aSelector] )
      return YES;

   // Der String carInfo kann die Nachricht handhaben
   if ([self.carInfo respondsToSelector:aSelector])
      return YES;

   // Anderenfalls...
   return NO;
}
```

```
// Erlaubt das Auftreten als Klasse
- (BOOL)isKindOfClass:(Class)aClass
{
   // Überprüft Car
   if (aClass == [Car class]) return YES;
   if ([super isKindOfClass:aClass]) return YES;

   // Überprüft NSString
   if ([self.carInfo isKindOfClass:aClass]) return YES;

   return NO;
}
```

Weiterleitung kinderleicht

Das Methodenpaar aus `methodSignatureForSelector:` und `forwardInvocation:` bildet eine bewährte Möglichkeit, um die Weiterleitung für Ihre Klassen einzusetzen. Auf dem iPhone ist auch ein einfacherer, aber weniger gut dokumentierter Ansatz möglich, den Sie auf eigene Gefahr verwenden können. Ersetzen Sie dazu die beiden Methoden durch eine einzige, die die gesamte Arbeit mit weniger Programmieraufwand verrichtet. Sie lässt sich bequem einsetzen, doch wenn Sie sich die Xcode-Dokumentation ansehen, werden Sie feststellen, dass sie auf dem iPhone offiziell nicht unterstützt wird.

```
- (id)forwardingTargetForSelector:(SEL)sel
{
   if ([self.carInfo respondsToSelector:sel]) return self.carInfo;
   return nil;
}
```

3.13 Zusammenfassung

In diesem Kapitel haben Sie eine abgekürzte, aber sehr informative Einführung in Objective-C und Foundation erhalten. Sie haben gelesen, wie Objective-C die Sprache C erweitert und Unterstützung für die objektorientierte Programmierung bietet. Sie haben Eigenschaften und die Speicherverwaltung kennengelernt und eine Übersicht über die wichtigsten Foundation-Klassen erhalten. Was können Sie nun aus diesem Kapitel mitnehmen? Merken Sie sich die folgenden abschließenden Gedanken:

- Der Beispielcode zu diesem Kapitel enthält alle Beispiele, die in dieser Einführung gegeben wurden. Versuchen Sie, diesen Code direkt in Xcode auszuprobieren. Spielen Sie damit herum, ergänzen Sie Ihre eigenen Codebeispiele, oder erweitern Sie die vorgestellten. Praktische Übungen sind die beste Möglichkeit, um wichtige Fertigkeiten für die iPhone-Entwicklung zu erlangen.

- Um Objective-C und Cocoa zu lernen, bedarf es mehr als nur der Lektüre eines Kapitels. Wenn Sie die Programmierung für das iPhone ernsthaft lernen wollen und Ihnen die hier vorgestellten Prinzipien neu sind, sollten Sie zu einem Buch greifen, das sich auf die Einführung dieser Technologien für Entwickler konzentriert, für die diese Plattform neu ist. Empfehlenswert sind beispielsweise *Cocoa: Programmierung für Mac OS X* von Aaron Hillegass, *Objective-C 2.0: Anwendungen entwickeln für Mac und iPhone* von Stephen Kochan und *Xcode 3* von Fritz Anderson.
- In diesem Kapitel wurden die Frameworks *Core Foundation* und *Carbon* erwähnt. Diese Technologien wurden aber nicht tiefschürfend behandelt. Allerdings werden Sie auf dem iPhone irgendwann APIs auf der Grundlage von C begegnen, vor allem dann, wenn Sie mit dem Adressbuch, mit Quartz-2D-Grafiken, mit Core Audio und anderen Frameworks arbeiten. Alle diese Bereiche sind auf der Entwicklerwebsite von Apple ausführlich und mit Codebeispielen dokumentiert. Eine sicherere Grundlage in der Programmierung mit C (und vielleicht auch C++) hilft Ihnen, mit den Einzelheiten der Implementierung fertigzuwerden.