FRAFORCE eBook

Speichermanagement in iOS

Franz Bruckhoff

http://fraforce.com

fraforce Academy http://fraforce.com

Speichermanagement in iOS

Version: 1.0.708

Letzte Änderung: Freitag, 28. Januar 2011

Statistik: 54 Seiten, 9778, Wörter, 70199 Zeichen

http://fraforce.de

DIN-A5

Inhalt

1 Wem gehört das Objekt?

1.1 Eigentum verpflichtet

Eine besondere Herausforderung im Speichermanagement ist in der Frage der Objekteigentümerschaft begründet. Was bedeutet es, Eigentümer zu sein? Ich erlaube mir an dieser Stelle einmal, ganz besonders weit her zu holen. Artikel 14 Absatz 2 des deutschen Grundgesetzes besagt: „Eigentum verpflichtet [...]“. Jeder ist also für sein Eigentum verantwortlich. Wir können unser schrottreifes Auto nicht einfach im Wald abstellen, sondern müssen es zum Schrottplatz bringen. Jeder muss seinen Müll ordentlich entsorgen, seine Kleidung waschen oder waschen lassen. Jeder ist für seine Handlungen selbst verantwortlich. Macht jemand die Küche schmutzig, muss er sie sauber machen. Nimmt jemand die letzte Rolle, muss er eine neue hinlegen. Belegt jemand irgendwo mit seinem Objekt Speicher, muss er den Speicher irgendwann wieder freigeben, und dafür sorgen, dass er damit seiner Umgebung keinen Schaden zufügt. Etwa ein Programmcrash mit dem unerfreulichen Fehlercode 101, oder die gefürchtete EXC_BAD_ACCESS Fehlermeldung.

Der Kern des Speichermanagements besteht aus einer Hand voll sehr einfachen Konventionen, an die Sie sich unbedingt halten müssen. Tun Sie das, werden Sie mit dem Speichermanagement kaum bis keine Probleme haben. Tun Sie das nicht, werden Ihnen Speicherprobleme und Fehlermeldungen wie EXC_BAD_ACCESS das Leben schwer machen.

Wir unterscheiden zwei Situationen im Speichermanagement: Entweder sind wir Eigentümer an einem Objekt und somit für dessen Speicher

verantwortlich, oder wir sind kein Eigentümer und somit für dessen Speicher nicht verantwortlich.

Ob wir für den Speicher verantwortlich sind oder nicht, können wir an Hand bestimmter Merkmale im Programmcode feststellen, welche per Konvention genau definiert sind. Manche Merkmale sind offensichtlich, während andere erst bei näherer Betrachtung identifizierbar sind.

Das offensichtlichste aller Merkmale dafür, das wir für den Speicher eines Objektes verantwortlich sind und somit Eigentümer am Objekt werden, ist die `+alloc` Nachricht an ein Klassenobjekt. Diese von `NSObject` geerbte Methode fordert von der Laufzeitumgebung Speicher an, der für eine Instanz der Klasse geeignet ist.

In zahlreichen Seminaren stellte ich fest, dass das Speichermanagement für Einsteiger sehr einfach zu meistern ist, sobald diese in der Lage sind genau zu erkennen ob eine Objekteigentümerschaft besteht oder nicht.

Die große Frage lautet also: Wer wird wann und warum Eigentümer des Objektes, und was genau heisst das?

Die Sache mit der Eigentümerschaft ist im Grunde genommen ganz einfach. Eigentümer wird der Scope (engl.: Geltungsbereich), in dem die Eigentümerschaft angefordert wurde.

Wir können die Eigentümerschaft auf unterschiedliche Weise anfordern:

- Fordern wir in einer Methode explizit Speicher an, ist diese Methode Eigentümer am neuen Objekt.
- Fordern wir in einem if-Block explizit Speicher an, so ist dieser if-Block Eigentümer am neuen Objekt.

- Weisen wir ein Objekt einer `retain`-Property zu, ist die Instanz Miteigentümer am Objekt.

Dies sind nur einige von vielen Möglichkeiten, Eigentümer an einem Objekt zu werden.

Was genau bedeutet es, Eigentümer zu sein? Welche Vorteile hat das? So lange ein Geltungsbereich, also z.B. ein if-Block oder eine Instanz Eigentümer an einem Objekt ist, wird der Speicher dieses Objektes nicht freigegeben. Der Eigentümer kann sich darauf verlassen, dass das Objekt so lange existiert, bis er es nicht mehr benötigt.

Aber: Eigentum verpflichtet auch. Der Eigentümer muss seine Eigentümerschaft am Ende seiner Existenz wieder abmelden. Was danach mit dem Objekt geschieht, interessiert den Eigentümer nicht. Vielleicht gibt es noch dutzende andere Miteigentümer. Vielleicht aber auch nicht. Das spielt für den jeweiligen Eigentümer keine Rolle. Das Ende der Existenz eines Eigentümers ist dann erreicht, wenn der Eigentümer nichts mehr tun kann. Ein if-Block ist zu Ende, wenn die letzte Zeile des Blocks abgearbeitet wurde. Eine Instanz hat das Ende ihrer Lebensdauer erreicht, wenn sie die `-dealloc` Nachricht erhält.

Wie Sie sehen ist der „Eigentümer“ unter Umständen etwas sehr abstraktes. So etwas wie ein if-Block, oder der Scope einer for-Schleife. Oder sogar eine Instanz, also ein Objekt.

Interessant finde ich, das es einfacher ist, die Eigentümerschaft mit dem Scope einer Methode oder eines if-Blocks zu erklären, als mit einer Instanz.

In einem if-Block ist der Fall eindeutig: Erzeugt der if-Block ein Objekt und gibt es nicht frei, hat er seine Eigentümerschaft nicht korrekt

abgemeldet. Probleme mit dem Speichermanagement sind dann vorprogrammiert.

Damit Sie das Speichermanagement beherrschen, ist Ihr Verständnis für die Objekteigentümerschaft und die damit einhergehende Verantwortung von großer Bedeutung. Sie müssen sich darüber völlig im Klaren sein, ob ein Scope Eigentümer eines Objektes ist, oder nicht.

Mit Scope meine ich den Bereich, in dem z.B. eine Variable gültig ist. Sei es eine Instanzvariable oder eine lokale Variable, die in einem if-Block deklariert wurde.

Die Aufgabe des Eigentümers ist es, unbedingt seine Eigentümerschaft irgendwann wieder aufzugeben. Tut ein Eigentümer das nicht und wird eliminiert, haben wir ein Memory Leak (engl.: Speicherleck). Der Eigentümer kann seine Eigentümerschaft nicht mehr aufgeben. Dies führt dann regelmäßig dazu, dass das Objekt bis zur Terminierung der Applikation im Speicher bleibt. Warum? Weil es angeblich noch einen Eigentümer gibt, der an dem Objekt interessiert ist.

Der Kern des Speichermanagements aus technischer Sicht ist das Reference Counting. Manchmal auch als Retain Counting bezeichnet. Dabei handelt es sich um ein System, bei dem jedes Objekt zählt wie viele Eigentümer es hat. Ein Objekt kann demnach nicht nur einen, sondern mehrere Eigentümer haben. So lange noch ein Eigentümer existiert, belegt das Objekt wertvollen Speicher. Die Laufzeitumgebung orientiert sich lediglich am Retain Count eines Objektes. Sobald dieser 0 ist, also sobald ein Objekt keinen Eigentümer mehr hat, ist dessen Speicher freigegeben. Das Objekt ist danach nicht mehr zu gebrauchen.

1.2 Das ungeschriebene Gesetz

Die Konventionen die für das fehlerfreie Speichermanagement geschaffen wurden sind sehr transparent und überschaubar. Orientieren Sie sich immer an diesem ungeschriebenen Gesetz. Bei strikter Einhaltung werden Sie mit dem Speichermanagement wenige bis keine Probleme haben.

§1 Ein Objekt kann einen oder mehrere Eigentümer haben.

§2 Hat ein Objekt keinen Eigentümer mehr, wird es von der Laufzeitumgebung aus dem Speicher entfernt.

§3 Ein Geltungsbereich kann Eigentümer oder Miteigentümer an einem Objekt werden. Ein Geltungsbereich gemäß §3 ist ein Bereich, in dem ein Bezeichner (eine Variable) gültig ist. Beispiele: Ein if-Block, eine Methode, oder sogar eine Instanz.

§4 Ein Geltungsbereich wird Eigentümer an einem Objekt, wenn er dafür eine Methode verwendet, deren Namen mit „new“ oder „`alloc`“ beginnt oder „copy“ enthält. Ein Geltungsbereich wird Miteigentümer an einem Objekt, wenn er an ein Objekt eine `-retain` Nachricht sendet.

§5 Jeder Eigentümer kann seine Eigentümerschaft an einem Objekt verstärken, in dem er sie mehrfach anfordert. Mehrfach verstärkte Eigentümerschaft muss entsprechend mehrfach abgemeldet werden.

§6 **Jeder Eigentümer muss seine Eigentümerschaft an einem Objekt zu einem geeigneten Zeitpunkt endgültig abmelden. Spätestens jedoch, wenn das Ende der Existenz des Eigentümers erreicht ist.** Siehe `-release`, `-autorelease`

§7 **Eine Methode oder Funktion, deren Name mit „new“ oder „`alloc`“ beginnt, oder deren Name „copy“ enthält, überträgt ihre Eigentümerschaft am erzeugten Objekt an den Aufrufer.**

§8 **Keine zyklischen Eigentümerschaften: In einem Objektgraphen sind Elternobjekte immer Eigentümer ihrer Kindobjekte. Kindobjekte sind jedoch nicht Eigentümer ihrer Elternobjekte.** Siehe „Strong Reference“, „Weak Reference"

Diese Konvention werden Sie besser verstehen, wenn Sie das Kapitel 2 über Reference Counting auf Seite 10 gelesen haben.

Das folgende Beispiel soll die Konventionen veranschaulichen:

```
- (void)setupDefaultChopper {
   // Fertigungsstätte für den Zusammenbau der Teile erzeugen:
   ChopperAssembly *chopperAssembly = [ChopperAssembly alloc];
   // „alloc“ = Achtung!
   chopperAssembly = [chopperAssembly init];

   // Rahmen erzeugen
   ChopperFrame *aluFrame = [ChopperFrame createDefaultFrame];
   // „create“ = Objekt ist „autoreleased“

   // Motor erzeugen:
```



```
int cylinders = 4;
Motor *motor = [Motor newMotorWithCylinders:cylinders];
// „new“ = Achtung!

// Befestigungsmaterial für den Motor holen
// Von aluFrame ein Objekt beziehen:
MotorMountings *mountings = [aluFrame mountingsForMotor:motor];

// mountings darf nicht verloren gehen: Eigentümer werden!
[mountings retain];

// Objekt erzeugen:
int spokes = 30; // Anzahl Speichen
Wheel *frontWheel = [Wheel wheelWithSpokes:spokes];
// +wheelWithSpokes: ist ein Convenience Constructor

// frontWheel darf nicht verloren gehen: Eigentümer werden!
[frontWheel retain];

// Teile montieren und foto rendern
Picture *picture;
NSString *modelName = @“Demo Cruiser“;
[chopperAssembly assembleForPicture:&picture // out-Parameter
                              frame:frame
                              motor:motor
                         frontWheel:frontWheel
                          modelName:modelName];

// picture ist jetzt initialisiert. Keine Eigentümerschaft.

UIImage *image = [picture image];
// image gehört diesem Geltungsbereich nicht.

// chopperView ist eine Property vom Typ UIImageView*
[self.chopperView setImage:image]; // gerendertes Bild anzeigen

[chopperAssembly release];
```

```
    [motor release];
    [mountings release];
    [frontWheel release];
    // aluFrame gehört nicht der Methode. Kein -release.
    // picture gehört nicht der Methode. Kein -release.
    // modelName gehört nicht der Methode. Kein -release.
    // image gehört nicht der Methode. Kein -release.
    // chopperView gehört nicht der Methode. Kein -release.
}
```

1.3 Über Eigentümer, Aktien und Aktionäre

Ein Objekt kann mehrere Eigentümer haben, und die Anteile dieser Eigentümer können ungleichmäßig verteilt sein. Ähnlich wie bei einer Aktiengesellschaft gibt es Kleinaktionäre und Großaktionäre. Ein Eigentümer kann seinen Anteil an einem Objekt also erhöhen oder verringern.

Allerdings hat ein Eigentümer beim Speichermanagement in der Regel keinen besonderen Vorteil davon, mehr als eine Aktie zu besitzen.

Mit jeder `-retain` Nachricht erwirbt ein Geltungsbereich eine Aktie und verstärkt somit seine Eigentümerschaft. Um die Eigentümerschaft wieder vollständig abzumelden, muss der Eigentümer alle Aktien zurück geben. Dies geschieht wahlweise mit einer `-release` Nachricht oder einer `-autorelease` Nachricht an das betroffene Objekt.

```
- (void)startMotor(Motor*)motor {
    [motor retain]; // Miteigentümer werden!
    [motor retain]; // Aktienanteil am Objekt erhöhen!
    [motor retain]; // Und nochmal erhöhen!
    [motor retain]; // Und weil es so viel spaß macht: Nochmal!
    // Damit ist -startMotor: nun vierfacher Eigentümer von motor

    [motor startWithAwesomeSoundEnabled:YES];
```

```
    // Da -startMotor: vierfacher Eigentümer ist, ist die Methode
    // auch vierfach für den Speicher des Objektes verantwortlich:
    [motor release];
    [motor release];
    [motor release];
    [motor release];
}
```

Da ein Objekt mehrere Eigentümer haben kann, darf ein einzelner Eigentümer niemals eine alleinige Entscheidung treffen, dass der Speicher eines Objektes freigegeben wird. So lange ein Objekt noch mindestens einen Eigentümer hat, wird der Speicher nicht freigegeben.

Ein ausgeklügeltes System sorgt dafür, das der Speicher eines Objektes freigegeben wird, sobald das Objekt keinen registrierten Eigentümer mehr hat. Mehr darüber erfahren Sie in Kapitel 2 „Reference Counting“ auf Seite 10.

Im folgenden Beispiel erzeugt eine Methode ein Objekt, und wird dadurch Eigentümer des Objektes:

```
- (void)startMotorWithCylinders:(int)cylinders {
    // „alloc“ = Achtung! Methode wird Eigentümer!
    Motor *motor = [Motor alloc];
    motor = [motor initWithCylinders:cylinders];
    [motor startWithSoundEnabled:YES];
    [motor release]; // Eigentümerschaft aufgeben
}
```

Wenn eine Methode Eigentümer eines Objektes geworden ist, muss sie ihre Eigentümerschaft am Objekt aufgeben. Dies geschieht spätestens am Ende der Methodenimplementierung. Entweder zeitverzögert per `-autorelease`, oder sofort per `-release`. In diesem Zusammenhang wird auch von „freigeben“ gesprochen.

Theoretisch könnte sich jede beliebige Methode um den Speicher eines beliebigen Objektes kümmern, auf das sie zugreifen kann. Theoretisch kann sich auch jede Person um die Angelegenheiten einer jeden Anderen kümmern. Aber das wäre das reinste Chaos.

2 Reference Counting

Beim Reference Counting (oft auch Retain Counting genannt) handelt es sich um ein Speichermanagementmodell. Ziel des Reference Counting ist es, den Speicher eines Objektes erst dann frei zu geben, wenn es keine Eigentümer mehr hat. Oder mit anderen Worten: Wenn es nicht mehr stark referenziert wird.

Grundsätzlich erben alle Klassen direkt oder indirekt von `NSObject`. `NSObject` implementiert die Methoden des Speichermanagements, darunter auch `+alloc`, `-retain`, `-release` und `-autorelease`.

2.1 Reference Count und Retain Count

Der reference count, oft auch retain count genannt, ist ein Zähler, der die Anzahl starker Referenzen auf ein Objekt zählt. Wir unterscheiden starke und schwache Referenzen. Eine starke Referenz auf ein Objekt besteht z.B. dann, wenn eine retain-Property ein Objekt referenziert. In diesem Fall sendet der Setter eine `-retain` Nachricht an das zugewiesene Objekt, gibt das bisher zugewiesene Objekt mit einer `-release` Nachricht frei, und weist das neue Objekt der entsprechenden Instanzvariable zu. Der Setter der retain-Property meldet also seine Eigentümerschaft am neu zugewiesenen Objekt an, und gibt seine Eigentümerschaft am zuvor referenzierten Objekt auf.

In Wirklichkeit werden also keine Referenzen gezählt, sondern die Anzahl der Eigentümerschaftsansprüche. Oder die Anzahl der „Aktien“, die insgesamt verteilt wurden.

Wenn z.B. eine Factory-Methode eine Klasse instanziiert, entsteht ein Objekt mit einem retain count von 1. Es gehört dann dem Scope, in dem das Objekt erzeugt wurde. Geschah dies in einem if-Block, gehört das Objekt dem if-Block. Der if-Block besitzt eine Aktie von diesem Objekt. Oft soll das Objekt über diesen Scope hinaus auf Instanzebene über einen längeren Zeitraum hinweg verwendet werden. Der Scope muss gemäß den Konventionen des Speichermanagements seine Eigentümerschaft an dem Objekt spätestens an seinem Ende seiner Existenz wieder aufgeben. Also spätestens in der letzten Zeile des if-Blocks. Die einzige Möglichkeit, wie das Objekt den Scope überleben kann, ist die Weitergabe an einen anderen Scope welcher Eigentümer wird. Im folgenden Beispiel übergibt eine Setup-Methode ein erzeugtes Objekt an die Instanz der Klasse, damit es auf Instanzebene verfügbar ist:

```
- (void)setupMotor {
   // „alloc“ = Achtung! Methode wird Eigentümer!
   Motor *motor = [[Motor alloc] init];
   self.engine = motor; // retain-Property sendet -retain an motor
   [motor release]; // Eigentümerschaft aufgeben
}
```

Für einen kurzen Moment hat das `motor` Objekt zwei Eigentümer: Die `-setupMotor` Methode und die Instanz, welche Empfänger der Nachricht `-setupMotor` ist. Erst nach dem die Instanz die Miteigentümerschaft am `motor` Objekt per `-retain` angemeldet hat, gibt die `-setupMotor` Methode ihre Eigentümerschaft am Objekt per `-release` auf. Diese Reihenfolge ist wichtig. Denn würde `-setupMotor` die Eigentümerschaft zu früh aufgeben, wäre der Speicher von `motor` schon freigegeben noch bevor

die Instanz die Chance hat eine Eigentümerschaft zu beanspruchen. Das Buffet wäre abgeräumt noch bevor es jemand sehen konnte. Schlimmer noch: Dem `-setEngine:` Setter würde ein Pointer mit einer korrupten Speicheradresse übergeben werden. Dieser würde dann eine `-retain` Nachricht an ein Objekt schicken, das nicht mehr existiert, was im besten Fall eine `EXC_BAD_ACCESS` Fehlermeldung zur Folge hätte.

Wird ein Objekt erzeugt, erhält es einen reference count von 1.

Mit Hilfe dieses Zählers ist das System in der Lage, den Speicher des Objektes erst dann freizugeben, wenn es keine starken Referenzen mehr auf das Objekt gibt. Oder wenn ein Objekt keinen Eigentümer mehr hat.

Benötigt ein Scope ein Objekt nicht mehr, an dem er eine Eigentümerschaft hat, muss er eine `-release` Nachricht an das Objekt schicken. Damit wird der reference count um 1 verringert.

Sobald der Reference Count eines Objektes 0 ist, wird automatisch eine `-dealloc` Nachricht an das Objekt geschickt, um den Speicher freizugeben. Die Implementierung von `-release` prüft mit `-retainCount` den aktuellen reference count des Objektes. Ist dieser 0, gibt sie den Speicher des Objektes frei. Wir rufen niemals selbst `-dealloc` auf, da wir somit das Reference Counting umgehen würden. Das wäre in etwa so, als würde jemand einfach das ganze erst kürzlich aufgebaute Buffet abräumen noch während sich die Leute daran bedienen.

2.2 Release Kaskade

Ziel des Reference Counting ist es, den Speicher eines Objektes erst dann freizugeben, wenn auf ein Objekt keine starken Referenzen mehr verweisen. Dies erkennt das System am Reference Count. Sobald dieser 0 ist, wird der Speicher freigegeben.

Auf Grund dieser Eigenschaft können mit einer `-release` Nachricht große Speicherfreigabewellen ausgelöst werden.

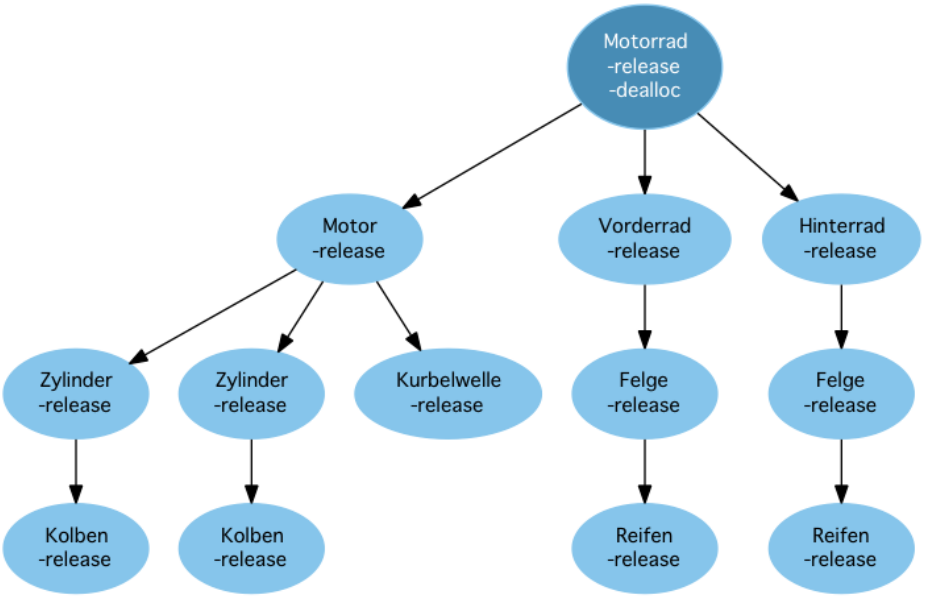
Ein Beispiel soll dies veranschaulichen. Stellen Sie sich eine Klasse vor, die für einen Motorrad-Konfigurator 3D-Motorräder baut. Sie erzeugen ein Motorrad-Objekt, und initialisieren es mit den Einstellungen des Benutzers. Anschließend erzeugt der Initializer des Motorrad-Objektes ein Hinterrad-Objekt, ein Vorderrad-Objekt und ein Motor-Objekt. Der Initializer des Motor-Objektes erzeugt ein Kurbelwellen-Objekt, sowie noch dutzende weitere Objekte, die zum Motor gehören. Auch die Vorder- und Hinterradobjekte erzeugen weitere Objekte.

Wenn die Konventionen bei der Entwicklung des Motorrad-Konfigurators eingehalten wurden, kann mit einer `-release` Nachricht an das Motorrad-Objekt eine Release Kaskade ausgelöst werden. Erreicht der reference count des Motorrad-Objektes 0, wird automatisch eine `-dealloc` Nachricht an das Motorrad-Objekt geschickt.

In der `-dealloc` Methode des Motorrad-Objektes werden `-release` Nachrichten an alle Objekte der nächst tieferen Ebene geschickt: An das Hinterrad-Objekt, an das Vorderrad-Objekt und an das Motor-Objekt. Da sonst niemand eine Eigentümerschaft an diesen Objekten hat, wird auch deren reference count 0, und sie erhalten eine `-release`

Nachricht. Die Release Kaskade setzt sich über die `-dealloc` Methoden der Objekte bis zur kleinsten Schraube fort.

Dies führt letztendlich zur Freigabe des Speichers aller Objekte, denen das Motorrad-Objekt direkt oder indirekt übergeordnet ist.



2.3 Strong Reference

Eine starke Referenz (strong reference) liegt vor, wenn ein referenziertes Objekt eine `-retain` Nachricht erhält. Damit wird eine Eigentümerschaft am Objekt zum Ausdruck gebracht. Der Speicher des stark referenzierten Objektes kann erst dann freigegeben werden, wenn die starke Referenz durch eine `-release` Nachricht an das Objekt aufgegeben wird.

Starke Referenzen kommen in den folgenden Szenarien zum Einsatz:

- In einer Methode wird ein Objekt aus einem Array geholt und per `-retain` stark referenziert. Im weiteren Verlauf werden weitere Methoden aufgerufen, die Zugriff auf dieses Array haben. Theoretisch könnten diese Methoden das Objekt aus dem Array entfernen, was jedoch außerhalb des Zuständigkeitsbereiches dieser Methode liegt. Da sie das Objekt bis zum Ende der Abarbeitung unbedingt benötigt, referenziert sie es stark. Am Ende gibt sie das Objekt wieder mit `-release` frei.
- Eine Instanzvariable eines Motorrad-Objektes hält eine Referenz auf ein Motor-Objekt, welches in einer `init`-Methode erzeugt und dann an den Setter der Instanzvariablen übergeben wurde. Da das Motor-Objekt für die Methoden des Motorrad-Objektes sehr wichtig ist, also über das Ende des Initializers hinaus benötigt wird, referenziert das Motorrad-Objekt das Motor-Objekt stark. Das Motorrad-Objekt ist damit Eigentümer am Motor-Objekt.

Im folgenden Beispiel wird eine retain-Property deklariert, welche eine starke Referenz erzeugt:

```
// Chopper.h
@interface Chopper : NSObject {
   Motor *motor;
}
@property(nonatomic, retain) Motor *motor;
@end
```

Im folgenden Beispiel erzeugt eine Methode eine starke Referenz auf ein Objekt, welches ihr übergeben wird:

```
- (void)washChopper:(Chopper*)chopper {
   [chopper retain]; // Der Chopper gehört nun auch dieser Methode
```

```
    // chopper aus dem Array entfernen
    // Gefahr: War vielleicht der einzige Eigentümer von chopper
    [self.dirtyChoppers removeObject:chopper];

    // chopper waschen

    // chopper ist sauber, und muss wieder freigegeben werden
    [chopper release];
}
```

Zugegeben, das Problem im obigen Beispiel lässt sich durch eine etwas intelligentere Lösung vermeiden. Das `chopper` Objekt könnte statt dessen erst nach dem Return der Methode aus dem Array entfernt werden. In den meisten Fällen läuft eine Methode nicht Gefahr, das ein an sie übergebenes Objekt während der Abarbeitung abhanden kommt.

2.4 Weak Reference

Eine schwache Referenz (weak reference) liegt vor, wenn ein Objekt lediglich referenziert wird. Ein prominentes Beispiel ist die assign-Property. Assign bedeutet „Zuweisung“. Im Gegensatz zu einer starken Referenz oder einer retain-Property wird bei einer schwachen Referenz auf eine Eigentümerschaft an dem Objekt verzichtet. Das Objekt erhält keine `-retain` Nachricht.

Eine schwache Referenz kommt z.B. dann zum Einsatz, wenn eine Wechselseitige Eigentümerschaft (siehe Kapitel 2 „Zyklische Eigentümerschaften“) vermieden werden soll. Auch Delegates werden schwach referenziert, um zyklische Eigentümerschaften zu vermeiden.

Beispiel: Ein Fotoalbum-Objekt erzeugt ein Foto-Objekt. Das Foto-Objekt soll Auskunft darüber geben können, zu welchem Fotoalbum es

gehört. Da das Fotoalbum-Objekt das Elternobjekt des Foto-Objektes ist, darf es das Fotoalbum-Objekt gemäß nicht stark referenzieren. Also erzeugt das Foto-Objekt lediglich eine schwache Referenz auf das Fotoalbum-Objekt.

Im folgenden Beispiel wird eine Instanzvariable mit einer schwachen Referenz auf ein Elternobjekt deklariert:

```
// Foto.h
@interface Foto : NSObject {
   FotoAlbum *album;
}
@property(nonatomic, assign) FotoAlbum *album;
@end
```

Im folgenden Beispiel wird einem Setter ein Objekt übergeben, welches einer Instanzvariablen zugewiesen werden soll. Die Methode verzichtet auf eine starke Referenz, da ein Elternobjekt übergeben wurde:

```
- (void)setParentAlbum:(Album*)newParentAlbum {
   if(newParentAlbum != parentAlbum) {
      parentAlbum = newParentAlbum; // schwache Referenz
   }
}
```

2.5 Zyklische Eigentümerschaften

Eine wechselseitige (zyklische) Eigentümerschaft liegt vor, wenn ein Objekt A eine Eigentümerschaft an einem Objekt B beansprucht, und das Objekt B eine Eigentümerschaft am Objekt A.

Beispiel: Ein Vorderrad gehört dem Motorrad. Das Motorrad gehört dem Vorderrad. Oder in Quellcode formuliert:

```
// Chopper.h
@interface Chopper : NSObject {
   Wheel *frontWheel;
```

```
}
@property(nonatomic, retain) Wheel *frontWheel;
@end

// Wheel.h
@interface Wheel : NSObject {
   Chopper *chopper;
}
@property(nonatomic, retain) Chopper *chopper;
@end
```

Gehen wir weiter davon aus, dass das Motorrad-Objekt irgendwo in einem Initializer das Vorderrad-Objekt erzeugt und dann die `frontWheel` sowie `chopper` Properties setzt. Der retain-Zyklus ist dann komplett.

Wo genau liegt das Problem bei solch einem retain-Zyklus? Der Übeltäter ist die `chopper` retain-Property der Klasse `Wheel`, welche sein Elternobjekt stark referenziert. Mit Elternobjekt meine ich dasjenige, welches das andere erzeugt hat. Das Motorrad-Objekt hat das Vorderrad-Objekt erzeugt, und ist somit sein Parent.

Da das Child bzw. Kindobjekt sein Elternobjekt stark referenziert, wird der retain count des Elternobjektes unter normalen Umständen niemals auf 0 sinken. Was passiert, wenn der letzte Eigentümer des Motorrad-Objektes das Motorrad-Objekt per `-release` frei gibt? Die Implementierung von -release prüft den `-retainCount`, und erst wenn dieser auf 0 sinkt sendet sie die `-dealloc` Nachricht an den Empfänger.

Betrachten wir nun die Implementierung von -dealloc in `Chopper`, der Klasse des Motorrad-Objektes:

```
- (void)dealloc {
   [frontWheel release], frontWheel = nil;
   [super dealloc];
}
```

Die Implementierung von `-dealloc` gibt also das Vorderrad-Objekt frei. Das Vorderrad-Objekt selbst wiederum würde in seiner `-dealloc` Implementierung das Motorrad-Objekt freigeben. Allerdings kommt es dazu nicht, da das Motorrad-Objekt die `-dealloc` Nachricht nicht erhält.

Zwickmühle. Eine fast hundertprozentige Garantie für ein Memory Leak. Die Einzige Möglichkeit die Objekte wieder los zu werden ist, dass das Vorderrad-Objekt sein Elternobjekt per `-release` freigibt. Und zwar noch lange bevor das Elternobjekt selbst freigegeben werden soll.

Die Lösung dieses Problems ist ganz einfach: Referenzieren Sie Elternobjekte und solche die es theoretisch sein könnten niemals stark.

Bei einem Motorrad und einem Vorderrad ist es ziemlich eindeutig. Das Motorrad ist das, was wir beim Motorradhändler kaufen. Das Vorderrad ist einfach ein Teil, welches zum Motorrad gehört. Es ist am Motorrad befestigt. Rein physikalisch betrachtet ist das Motorrad auch am Vorderrad befestigt, aber ich kenne niemanden der es wirklich so sieht.

Eine wechselseitige Referenz ist dennoch ohne weiteres möglich, wenn sich das Kindobjekt lediglich mit einer schwachen Referenz auf das Elternobjekt begnügt:

```
// Chopper.h
@interface Chopper : NSObject {
   Wheel *frontWheel;

}
@property(nonatomic, retain) Wheel *frontWheel;
@end

// Wheel.h
@interface Wheel : NSObject {
   Chopper *chopper;
}
@property(nonatomic, assign) Chopper *chopper;
@end
```

In einem anderen Szenario wäre es denkbar, das die `Chopper` Klasse nicht selbst eine Instanz der `Wheel` Klasse erzeugt und referenziert, sondern dass es sich um ein Baukastensystem handelt. Eine völlig andere Klasse könnte die beiden Instanzen erzeugen und

zusammenführen. Stellen Sie sich eine Factory Methode vor, welche die Teile erzeugt und zusammensetzt. Die Factory Methode ist Eigentümer dieser Objekte:

```
- (Chopper*)createChopper {
   Wheel *frontWheel = [[Wheel alloc] init];
   Wheel *rearWheel = [[Wheel alloc] init];

   Chopper *chopper = [Chopper alloc];
   [chopper initWithFrontWheel:frontWheel rearWheel:rearWheel];

   [frontWheel release];
   [rearWheel release];

   return [chopper autorelease];
}
```

Hier entsteht das gleiche Problem: Zwei Objekte referenzieren sich gegenseitig stark. Es ist unerheblich ob ein Objekt von einem anderen erzeugt wurde oder nicht. Interessant ist nur, ob es von einem anderen Objekt stark referenziert wird oder nicht. Also ob ein anderes Objekt Eigentümer ist oder nicht.

Nachfolgend ein paar weitere Beispiele, in denen eine eindeutige Parent-Child-Beziehung herrscht:

Eine Firma hat ein `NSArray` welches keinen, einen oder mehrere Mitarbeiter referenziert. Das Array gehört der Firma und stellt die Liste der Mitarbeiter dar. Die Mitarbeiter gehören der Liste. Streicht jemand einen Mitarbeiter von der Liste, ist er bei dieser Firma kein Mitarbeiter mehr. Wahrscheinlich existiert der Mitarbeiter aber weiterhin, weil er von anderen Objekten stark referenziert wird. Etwa die Stadt, bei der er als Einwohner gemeldet ist. Der Mitarbeiter referenziert die Firma in der er angestellt ist jedoch nur schwach. Sie gehört ihm nicht. Er

referenziert auch die Stadt in der er gemeldet ist nur schwach, also mit einer `assign`-Property. Die Stadt gehört ihm wahrscheinlich auch nicht.

Ein Motorrad kann ein neues Vorderrad bekommen. Ich habe selten davon gehört, das ein Vorderrad ein neues Motorrad bekommen hat. Natürlich passiert so etwas manchmal: Jemand findet im Gebüsch ein schönes Vorderrad, und macht sich dann auf die Suche nach einem dazu passenden Motorrad, um dieses Vorderrad fahren zu können. Schmarrn!

Eine besondere Situation stellen Delegate-Properties dar. Ein Delegate kann in der Regel jede beliebige Instanz sein. Theoretisch also auch diejenige, welche die Delegate-Property besitzt. Dies kommt z.B. dann vor, wenn wir eine Klasse spezialisieren die einen Delegate hat, und die Subklasse das Delegate-Protokoll implementiert und im Initializer die Delegate-Property auf `self` setzt. Delegates werden deshalb schwach referenziert.

3 Autorelease

Das Autorelease-System kommt immer dann zum Einsatz, wenn der Speicher eines Objektes nicht sofort freigegeben werden darf.

Es gibt aus Sicht des Speichermanagements zwei Arten von objekterstellenden Methoden, die ein Objekt zurück liefern. Die einen erstellen ein Objekt für eine andere Methode, ohne ihre Eigentümerschaft am Objekt abzumelden. Sie übertragen die Eigentümerschaft an den Aufrufer und sind der Konvention entsprechend benannt. Die Namen dieser Methoden beginnen mit

„alloc“, „new“ oder „copy“. Der Aufrufer muss sich um den Speicher des Objektes kümmern.

Die anderen sind etwas „bequemer“. Oder „convenient“. Sie erstellen für eine andere Methode ein Objekt, und übernehmen das Speichermanagement für das zurückgelieferte Objekt. Diese Methoden sind auf `-autorelease` angewiesen. Die Namen dieser Methoden beginnen typischerweise mit dem Namen der Klasse ohne Pseudonamespace-Präfix wie z.B. „`NS`“ bei `NSString`, oder mit „create“. Siehe Kapitel 5.1.2 „Convenience Constructor“, S. 38.

Das folgende Beispiel soll die Notwendigkeit einer verzögerten Speicherfreigabe verdeutlichen.

```
+ (Pizza*)pizzaWithTopping:(NSString*)strTopping {
   Topping *topping = [Topping alloc];
   topping = [topping initWithType:strTopping];

   Pizza *pizza = [[self alloc] initWithTopping:topping];

   [topping release]; // richtig!
   [pizza release]; // zu früh!

   return pizza; // pizza wurde schon „gegessen“. ist weg!
}
```

Noch bevor die fertige Pizza an den Aufrufer zurück gegeben wird, gibt die Implementierung von `-release` ihren Speicher frei, da ihr reference count auf 0 gesunken ist. Eine mögliche Lösung wäre, dass sich der Aufrufer um den Speicher der Pizza kümmern muss. Dies würde aber gegen die Speichermanagement-Konventionen verstoßen (siehe Kapitel 1.2 „Das ungeschriebene Gesetz“ auf S. 5). Der Aufrufer hat die Pizza nicht erstellt, sondern von einem Convenience Constructor erstellen lassen. Der Name der Methode beginnt nicht mit „`alloc`“, beginnt nicht

mit „new“ und enthält kein „copy“. Deshalb ist `+pizzaWithTopping:` dafür verantwortlich. Dies wäre auch der Fall gewesen, wäre der Name der Methode `-createPizzaWithTopping`.

Gäbe es `-autorelease` nicht, hätte `+pizzaWithTopping:` keine Chance, korrekt zu handeln. Das folgende Beispiel ist korrekt:

```
+ (Pizza*)pizzaWithTopping:(NSString*)strTopping {
   Topping *topping = [Topping alloc];
   topping = [topping initWithType:strTopping];

   Pizza *pizza = [[self alloc] initWithTopping:topping];

   [topping release]; // richtig!
   [pizza autorelease]; // richtig!

   return pizza; // pizza ist noch vorhanden. Lecker!
}
```

Die aufrufende Methode hat anschließend die Gelegenheit, mit `-retain` eine Eigentümerschaft an der Pizza anzumelden, bevor der Autorelease-Pool geleert wird. Meistens geschieht dies am Ende der Run Loop.

Beachten Sie, dass `orderedPizza` im folgenden Beispiel lediglich ein Pointer ist, der auf das von `+pizzaWithTopping:` erstellte Objekt verweist.

```
- (void)orderAndEatPizza:(NSString*)strTopping {
   Pizza* orderedPizza = [Pizza pizzaWithTopping:strTopping];
   // +orderAndEatPizza: ist nicht Eigentümer von orderedPizza

   [orderedPizza retain]; // Eigentümerschaft anmelden
   // +orderAndEatPizza: ist jetzt Eigentümer von orderedPizza

   int slices = 8;
   [orderedPizza cutIntoSlices:slices];
   [orderedPizza eatSlice];
```

```
   [orderedPizza release]; // Eigentümerschaft abmelden
}
```

Im folgenden Beispiel verzichtet die Methode auf die Anmeldung einer Eigentümerschaft und vertraut darauf, dass der Autorelease Pool erst später geleert wird. Meistens erst wenn alle Methoden abgearbeitet wurden und die Run Loop zu Ende ist.

```
- (void)orderAndEatPizza:(NSString*)strTopping {
   Pizza* orderedPizza = [Pizza pizzaWithTopping:strTopping];

   int slices = 8;
   [orderedPizza cutIntoSlices:slices];
   [orderedPizza eatSlice];
}
```

Ein anderes Beispiel: Stellen Sie sich einen Eisverkäufer vor, der Eiskugeln in Waffeln abfüllt. Das Eis gehört dem Verkäufer. Mit `-release` würde er das Eis sofort los lassen, sobald seine Hand gerade eben über das Territorium seiner Verkaufstheke heraus ragt. Es würde sofort runter fallen und wäre weg. Die Kunden würden nie zu ihrem Eis kommen. Es sei denn, der Eisverkäufer verwendet `-autorelease`. Dann würde er mit dem Loslassen noch eine weile warten.

So schön `-autorelease` auch sein mag: Die Münze hat meistens zwei Seiten. Stellen Sie sich vor, der Eisverkäufer lässt das Eis zwei Stunden lang nicht los. Die Kunden würden durchdrehen. Die gesamte Applikation „Eis verkaufen“ könnte in Schwierigkeiten geraten, da die Hände, mit denen er das Eis hält, knappe Ressourcen sind.

Wenn Sie `-autorelase` verwenden, müssen Sie sich immer darüber im Klaren sein, wie lange die Verzögerung der Freigabe andauert. Ansonsten kann es schnell passieren, das Sie zu viel Speicher

reservieren. Verwenden Sie das Autorelease-System nur in Fällen, in denen es wirklich nicht anders geht.

3.1 Autorelease Pool

Zweck eines Autorelease Pools ist es, den reference count eines oder mehrerer Objekte zu einem späteren Zeitpunkt um 1 zu verringern.

Autorelease Pools enthalten schwache Referenzen auf Objekte, die ihnen hinzugefügt wurden. Das bedeutet, dass beim Hinzufügen eines Objektes zu einem Autorelease Pool der reference count des Objektes nicht erhöht wird. Der Autorelease Pool wird also nicht Eigentümer des hinzugefügten Objektes.

Wird der Autorelease Pool freigegeben, sendet der Autorelease Pool eine `-release` Nachricht an alle referenzierten Objekte.

Stellen Sie sich einen Autorelease Pool wie eine Badewanne vor. Immer dann, wenn eine Methode ihre Eigentümerschaft an einem Objekt zeitverzögert aufgeben will, wirft sie eine schwache Referenz in die Badewanne. Stellen Sie sich die schwache Referenz auf das Objekt wie eine Kugel vor, die in der Badewanne schwimmt. Eine Kugel, in der die Speicheradresse des Objektes eingraviert ist. Da ein Objekt mehrere schwache Referenzen haben kann, können auch mehrere Kugeln in der Badewanne schwimmen. Irgendwann zieht jemand den Stöpsel der Badewanne:

```
[pool release];
// [pool drain] ist für Garbage Collection. Nicht auf dem iPhone.
// -drain verhält sich auf dem iPhone wie -release.
// Apple verwendet in der main()-Funktion -release.
```

Stellen Sie sich nun vor, dass Im Abflussrohr der Badewanne eine Vorrichtung integriert ist, welche die Speicheradresse der Kugel scannt und eine `-release` Nachricht an das referenzierte Objekt schickt. Für jede `-release` Nachricht wird der reference count des Objektes um 1 verringert. Sobald der reference count auf 0 sinkt, wird der Speicher des referenzierten Objektes freigegeben.

3.1.1 Vorhandene Autorelease Pools

In der main()-Funktion wird bereits ein Autorelease Pool im Hauptthread erstellt:

```
int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Innerhalb von `UIApplicationMain()` wird für jedes Event innerhalb der Event Loop am Anfang ein Autorelease Pool erzeugt und am Ende geleert.

Wenn der Benutzer auf einen Button klickt, und damit ein Event auslöst, wird sofort ein neuer Autorelease Pool erzeugt. Sobald das Event abgearbeitet ist, wird der Autorelease Pool geleert und damit zerstört.

So lange Sie nicht all zu viele Objekte erzeugen lassen, können Sie mit diesem Autorelease Pool auskommen. Lassen Sie z.B. innerhalb verschachtelter Schleifen sehr viele Objekte erstellen, die „autoreleased“ sind, können Sie dafür einen „näheren“ Autorelease Pool erstellen, der früher geleert wird.

3.1.2 Autorelease Pool erstellen und leeren

Technisch gesehen handelt es sich bei einem Autorelease Pool um eine Instanz der Klasse `NSAutoreleasePool`. Im folgenden Beispiel wird ein Autorelease Pool erzeugt, befüllt, und anschließend geleert.

```
- (Snowflake*)createSnowflake {
   Snowflake *snowflake = [[Snowflake alloc] init];
   [snowflake autorelease];
   return snowflake;
}

- (void)autoreleasePoolDemo {
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   NSUInteger i;
   for(i=0; i < 25; i++) {
      Snowflake *snowflake = [self createSnowflake];
      // Schneeflocke benutzen...
   }
   [pool release];
}
```

Wenn Sie einen Thread erzeugen und Cocoa Touch Klassen verwenden, müssen Sie einen Autorelease Pool anlegen, da Autorelease Pools Thread-spezifisch sind. Legen Sie keinen an, würden die zahlreichen `-autorelease` Nachrichten ins Leere gehen. Die Folge wären Speicherlecks.

3.1.3 Objekte zum Autorelease Pool hinzufügen

Die einfachste und bekannteste Möglichkeit ist folgende:

```
[anObject autorelease];
```


Erhält ein Objekt eine `-autorelease` Nachricht, wird eine schwache Referenz auf das Objekt in denjenigen Autorelease Pool gelegt, welcher zuletzt dem Autorelease Pool Stack des aktuellen Threads hinzugefügt wurde. Also in denjenigen, der ganz weit oben auf dem Stack liegt.

Obwohl es in der `NSAutoreleasePool` Klasse eine `-addObject:` Methode gibt, können Sie ein Objekt nicht gezielt einem bestimmten Autorelease Pool hinzufügen. Ein Beispiel soll dies verdeutlichen:

```
- (void)autoreleasePoolDemo {
   NSAutoreleasePool *poolA = [[NSAutoreleasePool alloc] init];
   NSAutoreleasePool *poolB = [[NSAutoreleasePool alloc] init];
   NSAutoreleasePool *poolC = [[NSAutoreleasePool alloc] init];

   NSString *string1 = [[NSString alloc] initWithCString:"str"];
   [string1 autorelease]; // wird zu poolC hinzugefügt

   NSString *string2 = [[NSString alloc] initWithCString:"str"];
   [poolB addObject:string2]; // wird zu poolC hinzugefügt

   [poolA release]; // leer
   [poolB release]; // leer
   [poolC release]; // string1 und string2 waren drin
}
```

Verwenden Sie immer `-autorelease` anstatt `-addObject:`.

3.2 Autorelease Pool Stack

In einer iPhone Applikation kann es mehrere Autorelease Pools geben. Wird ein weiterer Autorelease Pool erzeugt, landet dieser auf einem Stack. Sie können sich das bildlich in etwa so vorstellen:

Ihre Applikation hat einen Tellerstapel. Jedes mal, wenn ein Autorelease Pool erzeugt wird, wird dieser oben drauf gelegt. Und jedes mal, wenn

etwas in einen Autorelease Pool gelegt werden soll, landet es auf dem obersten Autorelease Pool. Also in denjenigen, der als letztes hinzugefügt wurde. Wird ein Autorelease Pool geleert, wird er vom Stapel runter genommen.

Wenn Sie also an ein Objekt eine `-autorelease` Nachricht schicken, wird eine schwache Referenz in den höchsten Autorelease Pool auf dem Autorelease Pool Stack gelegt. Sie legen also eine mit der Speicheradresse des Objektes gravierte Kugel in den Suppenteller der ganz oben auf dem Tellerstapel liegt. Und zwar in denjenigen des Threads, in dem Sie gerade operieren. Jeder Thread hat seinen eigenen Autorelease Pool Stack.

Das folgende Beispiel soll den Autorelease Pool Stack veranschaulichen:

```
// Chopper.m
+ (Chopper*)chopperWithCylinders:(int)cylNum {
   return [[[self alloc] initWithCylinders:cylNum] autorelease];
}

// Demo.m
- (void)autoreleasePoolDemo {
   NSAutoreleasePool *poolA = [[NSAutoreleasePool alloc] init];
   NSAutoreleasePool *poolB = [[NSAutoreleasePool alloc] init];
   NSAutoreleasePool *poolC = [[NSAutoreleasePool alloc] init];

   int cylinders = 2;
   Chopper *chopper = [Chopper chopperWithCylinders:cylinders];
   // Der Convencience Constructor verwendete -autorelease

   [poolA release]; // war leer
   [poolB release]; // war auch leer
   [poolC release]; // war was drin (chopper)
}
```


Sie können keine Referenz auf einen Autorelease Pool in einen Autorelease Pool legen.

Ein Autorelease Pool kann keine `-retain` Nachricht erhalten.

Autorelease Pool Stacks sind Thread-spezifisch. Jeder Thread hat seinen eigenen Autorelease Pool Stack.

Der Autorelease Pool Stack hat die folgenden Aufgaben:

- Steuerung, welchem Autorelease Pool eine Objektreferenz hinzugefügt werden soll, wenn ein Objekt eine `-autorelease` Nachricht erhält.
- Steuerung der Freigabe von Autorelease Pools, wenn ein Autorelease Pool freigegeben wird.

Wird ein Autorelease Pool freigegeben, der nicht zuletzt hinzugefügt wurde, also nicht ganz oben auf dem Stack liegt, so werden alle später hinzugefügten Autorelease Pools ebenfalls freigegeben. Dieses Verhalten ist Thread-spezifisch und bezieht sich nur auf Autorelease Pools, die auf dem gleichen Autorelease Pool Stack liegen. Betrachten Sie dies nicht als ein „Feature“, sondern vielmehr als ein „Notfallsystem“. Es ist z.B. dann nützlich, wenn Sie es einmal versäumt haben sollten, einen Autorelease Pool freizugeben.

Grundsätzlich gibt es zwei Situationen, in denen wir einen Autorelease Pool erzeugen wollen: Entweder, weil es noch keinen Autorelease Pool auf dem Autorelease Pool Stack gibt (z.B. Einstiegspunkt in einen neuen Thread), oder weil wir einen näheren Autorelease Pool in einer Schleife benötigen, welche in mehreren tausend Durchläufen vielleicht hunderttausende per `-autorelease` erzeugte Objekte produziert. Oft insbesondere `NSString` Instanzen, die per Convenicence Konstruktoren erzeugt werden. Nutzen wir in solch einer Schleife Framework-Methoden

deren Implementierung außerhalb unseres Zuständigkeitsbereiches liegt, müssen wir regelmäßig davon ausgehen, das dort per `-autorelease` freigegebene Objekte erzeugt werden. Diesen „nahen“ Autorelease Pool erzeugen wir bei jedem neuen Schleifendurchlauf und geben ihn nach jedem Ende wieder frei. Manchmal z.B. auch nach allen 10 durchläufen, da `+alloc` Operationen teuer sind.

4 Die Speichermanagement-Methoden von NSObject

4.1 +alloc

„Alloc“ steht für „allocate“ und bedeutet „zuteilen“, „reservieren“, „belegen“.

Diese Klassenmethode reserviert den notwendigen Speicher zur Erzeugung eines Objektes und liefert an den Aufrufer eine Instanz der Klasse zurück. Das zurück gelieferte Objekt entspricht dabei einer Datenstruktur, welche die Klasse beschreibt, welche die `+alloc` Nachricht erhalten hat. Der retain count des zurück gelieferten Objektes ist 1.

Die `+allocWithZone:` Methode können Sie momentan ignorieren, da iOS zum Zeitpunkt dieses Schreibens (28. Januar 2011) nicht über eine Auslagerungsdatei (Swap File) verfügt. Eine Auslagerungsdatei vergrößert virtuell den Adressraum des Arbeitsspeichers, und dient dazu momentan nicht benötigte Daten im Speicher auf die Festplatte durch Paging bzw. Swapping auszulagern um Platz für wichtigere Dinge zu schaffen. Dieser Paging- bzw. Swapping-Prozess ist teuer und kann zu Performancebeinträchtigungen führen. In bestimmten Situationen

kann es zu einer Überlastung führen, also zum sogenannten Thrashing kommen. Dies geschieht z.B. dann, wenn Objekte in einem Objektgraphen intensiv miteinander kommunizieren und ein Teil dieses Objektgraphen ausgelagert wird. Das Betriebssystem muss dann jedes mal, wenn ein ausgelagerter Teil benötigt wird, diesen ausgelagerten Teil in den Arbeitsspeicher laden. Wenn es sehr unglücklich verläuft, lagert es dafür den anderen gleich im Anschluss benötigten Teil aus, um Platz zu schaffen. `+allocWithZone:` erlaubt es eine Speicherzone anzugeben. Damit können Entwickler dazu beitragen, das Objekte die viel miteinander kommunizieren oder zusammengehören im Speicher nah beieinander liegen. Damit verringern Sie die Wahrscheinlichkeint einer Überlastung. In diesem Zusammenhang ist die `-zone` Methode interessant. Diese gibt die Speicherzone eines Objektes zurück, welche als Argument an `+allocWitzZone:` übergeben werden kann. Sie können damit bereits jetzt zukunftssicher entwickeln. Allerdings halte ich es nicht für erforderlich. Ob Auslagerungsdateien in den nächsten Jahren auf mobilen iOS Geräten relevant werden, wage ich zu bezweifeln. Unter iOS wird immer die `NSDefaultMallocZone` verwendet.

Wir beschränken uns also bis auf Weiteres auf `+alloc`.

In der Praxis sieht dies so aus:

```
*Motor motor = [Motor alloc];
```

Per Konvention folgt auf ein `+alloc` immer der Aufruf eines sogenannten Initializers. Standardmäßig beginnt dieser mit `-init`. Der Initializer initialisiert die Instanzvariablen des Objektes, legt also deren „Startwerte“ fest:

```
*Motor motor = [[Motor alloc] init];
```

Verwenden Sie `+alloc` und `-init` immer paarweise. Die beiden gehören zusammen wie Fische und Wasser. Oft werden Ihnen passendere -init... Methoden geboten, so wie in diesem Beispiel:

```
NSString *assemblyCode = @"DE12";
int assemblyYear = 2009;
int assemblyQuarter = 2;
NSString *chopperSerial = [NSString alloc];
chopperSerial = [chopperSerial initWithFormat:@"%@-%d%d",
assemblyCode, assemblyYear, assemblyQuarter];
```

4.2 -retain

„Retain“ bedeutet „einbehalten“. Wird diese Nachricht an ein Objekt geschickt, wird dessen reference count um 1 erhöht. Damit signalisiert die aufrufende Methode die Eigentümerschaft an dem Objekt.

Im folgenden Beispiel wird ein Setter für eine Instanzvariable implementiert. Oft ist es so, dass die Instanzvariable ein Objekt dauerhaft referenzieren soll. In diesem fall sendet die Methode an das Objekt, welches ihr übergeben wurde, eine `-retain` Nachricht. Damit ist sichergestellt, dass der Speicher des Objektes frühestens dann freigegeben wird, wenn sich niemand mehr für das Objekt interessiert.

```
- (void)setMotor:(Motor*)newMotor {
   if(newMotor != motor) { // nicht das gleiche Objekt?
      [motor release]; // altes Objekt freigeben (§6)
      motor = [newMotor retain]; // neues Objekt halten
   }
}
```

Senden Sie `-retain` niemals an einen Autorelease Pool.

4.3 -release

„Release“ bedeutet „freigeben“, „loslassen“. Wird diese Nachricht an ein Objekt geschickt, wird damit die Aufgabe der Eigentümerschaft zum Ausdruck gebracht. Der retain count wird um 1 verringert.

Im folgenden Beispiel erzeugt eine Methode ein Objekt, verwendet es, und gibt es anschließend frei.

```
- (void)makeMilkShake {
   Milk *freshMilk = [[Milk alloc] initWithCow:self.cow];
   [freshMilk shake];

   // trinken...

   [freshMilk release]; // danke, wird nicht mehr gebraucht!
}
```

4.4 -autorelease

„Autorelease“ bedeutet „automatisch freigeben“, „automatisch loslassen“. Wird diese Nachricht an ein Objekt geschickt, so wird es im Autorelease Pool für die spätere Freigabe registriert. Dies ist immer dann sinnvoll, wenn eine Methode die Eigentümerschaft an einem Objekt zwar aufgeben will, aber nicht sofort aufgeben darf.

Das folgende Beispiel zeigt ein valides Szenario für die Verwendung von `-autorelease`. Ein Convenience Konstruktor erzeugt für seinen Aufrufer ein Objekt. Da die erzeugende Methode gemäß Konvention Eigentümer des Objektes ist, muss sie dessen Speicher freigeben, sobald sie es nicht mehr benötigt. Da der Convenience Konstruktor nach Return des Objektes keinen Zugriff mehr darauf hat, registriert er das Objekt in einem Autorelease Pool, der zu gegebener Zeit geleert wird. Oft erst

wenn die Run Loop zu Ende ist, also wenn alle Methoden abgearbeitet wurden.

```
+ (Motor*)motorWithHorsePower:(int)horsePower {
   return [[[self alloc] initWithHPower:horsePower] autorelease];
}
```

Melden weitere Methoden die Eigentümerschaft an dem Objekt an, wird dessen Speicher durch das Autorelease nicht vorzeitig freigegeben.

Das Autorelease funktioniert nur in Kombination mit einem Autorelease Pool. Wann immer möglich, sollten Sie auf `-autorelease` verzichten, und den Speicher des Objektes mit `-release` freigeben.

Die verzögerte Freigabe kann z.B. in Schleifen dazu führen, dass der Speicher zu schnell aufgebraucht wird und die automatisch verzögerte Freigabe zu spät erfolgt. Die Folge wäre eine Low Memory Warning bis hin zur Terminierung des Programms auf Grund des zu hohen Speicherverbrauchs.

Senden Sie `-autorelease` nicht an einen Autorelease Pool.

4.5 -dealloc

„Dealloc“ steht für „deallocate“ und bedeutet „Zuteilung aufheben“, „freigeben“. Diese Methode wird in der Implementierung von `-release` automatisch aufgerufen, sobald der retain count eines Objektes 0 ist.

Genau genommen handelt es sich dabei um eine Template-Methode, welche Sie überschreiben um dort alle starken Referenzen aufzugeben welche Sie auf Instanzebene erzeugt haben. `-dealloc` dient jedoch nicht

ausschließlich diesem Zweck. Meldet sich ein Objekt irgendwo selbst als Delegate an, sollte es sich spätestens in `-dealloc` wieder als Delegate abmelden, sofern das Objekt welches die Delegate-Property besitzt noch existiert. Registriert sich ein Objekt in einem Initializer als Observer für Property (KVO bzw. Key-Value Observing) oder als Empfänger für Notifications bei einem Notification Center, muss es sich spätestens in `-dealloc` wieder abmelden.

Darüber hinaus müssen Sie einige Besonderheiten beachten. Speziell bei UIViewController. Eine retain-Property muss nicht zwangsläufig immer in `-dealloc` freigegeben werden. Es kommt darauf an, an welcher Stelle die starke Referenz erzeugt wurde. Erzeugen Sie diese in `-loadView:` oder `-viewDidLoad:`, geben Sie die starke Referenz in `-viewDidUnload:` frei, da ein View Controller sein View freigeben und wieder neu laden kann. Würden Sie nichts desto trotz alles in `-dealloc` freigeben, würden die Bemühungen des View Life Cycle Managements nur wenig Früchte tragen. Die stark referenzierten Objekte würden wie etwa eine schwerlastige programmatisch in `-viewDidLoad:` erzeugte UIWebView Instanz im Speicher bleiben, und erst nach Speicherfreigabe des View Controllers in seiner `-dealloc` Methode freigegeben werden. Unter Umständen viel zu spät.

Darüber hinaus wird die `-dealloc` Nachricht nicht zwingend an ein Objekt gesendet, wenn eine Applikation terminiert wird. Das Betriebssystem holt sich einfach den gesamten belegten Speicher der Applikation zurück, was wesentlich schneller geht.

Am Ende leiten Sie die -dealloc Nachricht immer an `super` weiter, damit die Superklasse ebenfalls ihrer Pflicht nachkommen kann und z.B. starke Referenzen aufgibt. Ganz nach dem Motto: Jeder räumt sein eigenes Zeug weg.

Senden Sie `-dealloc` nicht manuell. Dies geschieht automatisch. Ausnahme: `-dealloc` an `super`.

Die `-dealloc` Methode gibt alle vom Objekt belegten Ressourcen frei. In der Praxis bedeutet das, dass die `-dealloc` Methode die Eigentümerschaften an Objekten aufgibt, welche von Instanzvariablen referenziert werden. Dies ist der primäre Zweck dieser Methode.

Das folgende Beispiel zeigt eine typische Implementierung einer `-dealloc` Methode:

```
- (void)dealloc {
   [motor release];
   [frontWheel release];
   [rearWheel release];
   [chopperFrame release];

   [super release]; // Muss!
}
```

4.6 -copy

Die `-copy` Methode erzeugt ein neues Objekt auf Basis eines bestehenden Objektes. Das zurück gelieferte Objekt erhält einen reference count von 1. Aus Sicht der Speichermanagement-Konventionen können Sie `-copy` wie `+alloc` betrachten. Der aufrufende Scope wird Eigentümer.

Eine `-copy` Nachricht wird immer dann an ein Objekt gesendet, wenn es unerwünscht ist, dass ein möglicher Eigentümer des Objektes daran etwas verändert.

Das folgende Beispiel soll dies veranschaulichen:

```
- (void)analyzeWeather:(Weather*)currentWeather {
   // Von den aktuellen Wetterdaten wird eine Kopie gemacht
   Weather *currentWeatherSnapshot = [currentWeather copy];

   // Der Wetterdaten-Snapshot wird 5 Sekunden lang analysiert,
   // während sich currentWeather ständig verändert.
}
```

Die Wetterstation, welche fortlaufend ein `currentWeather` Objekt mit umfangreichen Messdaten aktualisiert, übergibt das `currentWeather` Objekt an die `-analyzeWeather:` Methode. Bevor diese mit der Analyse beginnt, kopiert sie das Objekt, um einen Schnappschuss vom aktuellen Zustand zu haben. Dieses Objekt kann dann zusammen mit der Wetterdatenanalyse archiviert werden.

Wenn Sie ein Objekt kopieren, ist es so, als wenn Sie ein Objekt neu erstellen. Die `-copy` Methode erzeugt ein neues Objekt mit einem retain count von 1. Anschließend gibt sie das neue Objekt an den Sender der Nachricht zurück. Anders als beim Aufruf eines Convenience Konstruktors, muss sich der Sender der `-copy` Nachricht um den Speicher des neuen Objektes kümmern.

5 Best Practice

5.1 Instanziierung und Initialisierung

Glücklicherweise gibt es einige sehr hilfreiche Konventionen. Bei strikter Einhaltung dieser Konventionen minimieren Sie die Gefahr von Speicherlecks. Schauen Sie sich ggf. noch einmal Kapitel 1.2 „Das ungeschriebene Gesetz“ auf Seite 5 an.

5.1.1 Object Factories

Eine Object Factory (Objektfabrik) ist eine Methode, die ein konfiguriertes Objekt erzeugt und an den Aufrufer zurück gibt. So lange der Name der Methode nicht mit „new“ oder „`alloc`“ beginnt, und kein „copy“ enthält, muss sich der Aufrufer nicht um den Speicher des erzeugten Objektes kümmern. Der Aufrufer geht der Konvention entsprechend davon aus, dass er den Speicher des Objektes, welches die Object Factory erzeugt, nicht freigeben muss. Die Object Factory muss an das erzeugte Objekt eine `-autorelease` Nachricht schicken, bevor es an den Aufrufer zurückgegeben wird. Für den Anwender verhält es sich so wie bei einem Convenience Constructor. Eine Object Factory erkennen Sie entweder an einem „create“ im Methodennamen, oder daran, das sich der Name der Klasse des zu erzeugenden Objektes im Methodennamen der Object Factory befindet.

Beispiele:

```
- (Pizza*)pizzaWithTopping:(*Topping)topping;
- (RoomLight*)roomLightWithWatts:(NSInteger)watts;
- (Pizza*)createPizza;
- (RoomLight*)createRoomLightWithWatts:(NSInteger)watts;
```

5.1.2 Convenience Constructor

Ein Convenience Constructor (engl.: „Bequemlichkeitskonstruktor“) erzeugt eine initialisierte Instanz und kümmert sich um den Speicher des erzeugten Objektes. Der Aufrufer kann das Objekt benutzen und vergessen, ohne sich um den Speicher zu kümmern. Mit anderen Worten: Der Aufrufer wird nicht Eigentümer des Objektes.

Einer der beliebtesten Konvenience Konstruktoren verbirgt sich hinter der Verwendung eines `NSString` Literals, welches in Wahrheit eine per `-autorelease` freigegebene `NSString` Instanz erzeugt:

```
NSString *str = @"Hello World!";
```

Sie müssen sich also nicht um die Freigabe des Speichers kümmern. Erstellen Sie jedoch einen Convenience Konstruktor, müssen Sie den Speicher des erzeugten Objektes zeitverzögert per `-autorelease` freigeben. So erhält der Aufrufer die Gelegenheit, eine Eigentümerschaft per `-retain` zu beanspruchen, bevor der Speicher des Objektes nach freigabe des Autorelease Pools freigegeben wird.

Warum ist das Autorelease absolut notwendig? Im Falle eines einfachen `-release` an das erzeugte Objekt würde der retain count des Objektes auf 0 sinken, noch bevor der Aufrufer die Chance erhält es zu benutzen oder sogar eine Eigentümerschaft per `-retain` zu beanspruchen. Durch das -autorelease übergibt der Convenience Konstruktor seine Eigentümerschaft an den Autorelease Pool ab, welcher meistens am Ende der Run Loop freigegeben wird.

Das Grundgerüst eines Convenience Konstruktors sieht so aus:

```
+ (Motor*)motorWithHorsePower:(int)horsePower {
   return [[[self alloc] initWithHPower:horsePower] autorelease];
}
```

Gemäß Namenskonvention haben Sie es immer dann mit einem Convenience Konstruktor zu tun, wenn der Name der Klassenmethode zur Instanziierung und Initialisierung dem folgenden Muster entspricht:

```
Motor *motor = [Motor motorWithHorsePower:horsePower];
```

Der Convenience Constructor der Klasse `Motor` enthält als ersten Namensbestandteil den Namen der Klasse mit kleinem Anfangsbuchstaben.

Bei dieser Konvention gibt es noch eine Sache, die zu beachten ist. Auf Grund des fehlenden Supports für Namensräume haben viele Klassen ein

Präfix. So etwa die NextStep-Klassen des Foundation Frameworks, deren Präfix NS lautet. In diesen Fällen wird das Präfix im Methodennamen des Convenience Konstruktors weg gelassen:

```
NSString *greeting = [NSString stringWithFormat:"Hi %@", name];
```

Trotz der Klasse `NSString` lautet der Methodenname des Convenience Constructors `+stringWithFormat:`, und nicht etwa `+nSStringWithFormat:` oder schlimmer noch: `+NSStringWithFormat:`.

5.1.3 +alloc und Initializer

Immer dann, wenn Sie ein Objekt durch `+alloc` erzeugen, müssen Sie sich um den Speicher kümmern. Dabei ist es unerheblich ob Sie das Objekt im nächsten Schritt auch initialisieren, oder nicht. Foundation baut auf das Two-Stage Creation Pattern auf, welches die Speicherallozierung und Initialisierung auf zwei voneinander getrennte Schritte aufteilt. Relevant ist hier nur die Speicherallozierung, also der erste Schritt. Ob Sie Ihr Objekt initialisieren oder nicht ist für die Speicherverwaltung irrelevant. Das Objekt nicht zu initialisieren verschafft Ihnen i.d.R. zahlreiche andere, oft weitaus gravierendere Probleme als Speicherlecks.

```
- (void)startMotorWithCylinders:(int)cylinders {
   // „alloc“ = Achtung! Methode wird Eigentümer!
   Motor *motor = [Motor alloc];
   motor = [motor initWithCylinders:cylinders];
   [motor startWithAwesomeSoundEnabled:YES];
   [motor release]; // Eigentümerschaft aufgeben
}
```

5.2 Accesoren

5.2.1 Getter

Der Aufrufer des Getters ist nicht für den Speicher verantwortlich. Im folgenden Beispiel gibt der Getter lediglich die Instanzvariable zurück:

```
- (Wheel*)frontWheel {
   return frontWheel;
}
```

Je nachdem, wie der Setter implementiert ist, kann es sein, dass dieser vor der Zuweisung eines neuen Objektes das alte Objekt aufgibt, und an das neue Objekt eine `-retain` Nachricht schickt. Beispiel:

```
- (void)setFrontWheel:(Wheel*)newFrontWheel {
   if(newFrontWheel != frontWheel) { // nicht das gleiche Objekt?
      [frontWhel release]; // altes Objekt freigeben
      frontWheel = [newFrontWheel retain]; // neues Objekt halten
   }
}
```

5.2.2 Setter

Einem Setter wird ein Objekt übergeben. Die meisten Setter weisen das Objekt direkt einer Instanzvariablen zu. Andere nutzen es als Berechnungsgrundlage um Instanzvariablen mit entsprechenden Werten zu setzen. Der `-setFrame:` Setter von `UIView` nimmt z.B. ein `CGRect` entgegen und setzt darauf basierend die `center` und `bounds` Properties.

Da ein Setter das ihm übergebene Objekt nicht selbst erzeugt hat, gehört es ihm nicht. Oft muss ein Objekt stark referenziert werden. Wir sprechen dann von einem retain-Setter oder von einer retain-Property, die einen entsprechenden retain-Setter hat.

Der retain-Setter prüft zunächst, ob das neue Objekt nicht mit dem alten Objekt identisch ist. Also ob die Speicheradressen unterschiedlich sind. Handelt es sich tatsächlich um unterschiedliche Objekte, erhält das alte Objekt eine `-release` Nachricht, und das neue Objekt eine `-retain` Nachricht. Anschließend weist der Setter das neue Objekt der Instanzvariablen zu.

Wenn Sie einen Setter selbst implementieren müssen, verwenden Sie das folgende Grundgerüst:

```
- (void)setFrontWheel:(Wheel*)newFrontWheel {
   if(newFrontWheel != frontWheel) { // nicht das gleiche Objekt?
      [frontWhel release]; // altes Objekt freigeben
      frontWheel = [newFrontWheel retain]; // neues Objekt halten
   }
}
```

Beachten Sie, das die automatisch generierten Setter die Sie mit `@synthesize` erzeugen wesentlich mehr tun als hier dargestellt. So ermöglichen diese z.B. das Key-Value Observing durch die Versendung bestimmter Nachrichten vor und nach der Änderung der Instanzvariablen und fügen Code für die Threadsicherheit der Instanzvariablen ein.

Bei copy-Settern gibt es eine besondere Ausnahmeregel. Ein copy-Setter darf ein neues Objekt erzeugen und Eigentümer werden, ohne seine Eigentümerschaft per `-release` oder `-autorelease` aufzugeben. Statt dessen überträgt der copy-Setter die Eigentümerschaft an die Instanz, welche ihre Eigentümerschaft wiederum in `-dealloc` per `-release` aufgibt. In bestimmten Ausnahmefällen ist `-dealloc` dafür ungeeignet. Das View Life Cycle Management von UIViewController sieht z.B. vor, das retain- oder Copy-Properties die in `-loadView` oder

`-viewDidLoad` gesetzt werden, nicht in `-dealloc` freigegeben werden, sondern in `-viewDidUnload`.

5.2.3 Automatische Getter und Setter

Lassen Sie Getter und Setter wann immer möglich automatisch erzeugen. Im folgenden Beispiel weisen wir den Compiler an, Getter und retain-Setter für eine Instanzvariable zu erzeugen:

```
// Chopper.h
@interface Chopper : NSObject {
   Wheel *frontWheel;
}
@property(nonatomic, retain) Wheel *frontWheel;
@end

// Chopper.m
@implementation Chopper
@synthesize frontWheel; // Erzeugt automatisch Getter und Setter
```

Die `@property` Compiler Direktive weist den Compiler an, Getter und Setter im Header bekannt zu machen.

Die `@synthesize` Compiler Direktive weist den Compiler an, Getter und Setter zu implementieren, sofern diese nicht implementiert sind. Das, was damit erzeugt wird, können Sie im Quelltext nicht sehen. Allerdings können Sie die Getter und Setter überschreiben. Setzen Sie wann immer möglich die Compiler Direktiven für Getter und Setter ein.

Der Code, der unsichtbar für Sie generiert wird, sieht in der Implementierung vereinfacht entsprechend so aus:

```
- (Wheel*)frontWheel {
   return frontWheel;
}
```

```
- (void)setFrontWheel:(Wheel*)newFrontWheel {
   if(newFrontWheel != frontWheel) {
      [frontWhel release];
      frontWheel = [newFrontWheel retain];
   }
}
```

In Wahrheit steckt hier noch ein wenig mehr Code für KVO und manchmal auch für Threadsicherheit drin. Sie können auf die Identitätsprüfung der Objekte verzichten, wenn Sie die Reihenfolge der Nachrichten ändern:

```
- (void)setFrontWheel:(Wheel*)newFrontWheel {
   [newFrontWheel retain];
   [frontWhel release];
   frontWheel = newFrontWheel;
}
```

Bei einem assign-Setter hingegen interessiert Sie der Speicher nicht:

```
- (void)setChopper:(Chopper*)newChopper {
   chopper = newChopper;
}
```

Ein copy-Setter verhält sich wie ein retain-Setter. Der copy-Setter kopiert das übergebene Objekt und wird dadurch Eigentümer des neu erstellten Objektes. Das alte Objekt wird freigegeben, und das neu erstellte Objekt wird der Instanzvariablen zugewiesen.

5.3 Outlets

Outlets werden gesetzt, wenn der Nib Loader die eingefrorenen Instanzen in den Speicher lädt.

In den meisten Fällen müssen Sie sich um die Freigabe des Speichers kümmern. Es hängt davon ab, ob Sie für Ihre Outlets retain- oder

assign-Properties benutzen. Bei einer assign-Property müssen Sie sich nicht um den Speicher kümmern.

```
@property (***, assign) ***
```

Eine assign-Property ist jedoch gefährlich, wenn Sie in Interface Builder ein Controller Objekt wie z.B. einen View Controller oder einen Navigation Controller eingefroren haben. Das View eines View Controllers wird als Subview in die View Hierarchie eingefügt und somit automatisch stark referenziert wird. Das bedeutet aber nicht, das auch der View Controller selbst stark referenziert wird. Das View bleibt erhalten. Das View Controller Objekt hingegen läuft Gefahr aus dem Speicher entfernt zu werden, wenn Sie es keinem Outlet zuweisen oder einem Outlet welches lediglich eine assign-Property ist.

Views die Sie in Interface Builder als Subview zum View eines View Controllers hinzufügen, müssen nicht zwingend per retain-Property referenziert werden. Oft ist nicht einmal ein Outlet notwendig. Wenn Sie z.B. einen UIButton als Subview hinzufügen und diesen programmatisch ansteuern wollen, reicht ein Outlet mit einer assign-Property für den Button aus, da der Button selbst von der View Hierarchie stark referenziert wird:

```
@property (nonatomic, assign) UIButton *myButton;
```

Wenn der View Controller im Rahmen seines View Life Cycle Managements sein View von seinem Superview befreit und es anschließend per `-release` freigibt, und das View keinen weiteren Eigentümer hat, gibt die Implementierung von `-release` dessen Speicher frei. Der Button verschwindet dann ebenfalls automatisch, da er keinen weiteren Eigentümer mehr hat.

Lädt der View Controller sein zuvor verworfenes View wieder in `-loadView`, tritt das Nib Loading System wieder in Aktion und setzt die Outlets neu.

Verwenden Sie statt dessen ein Outlet mit einer retain-Property, geben Sie den Button in `-viewDidUnload:` frei.

Wenn Ihr IBOutlet also „assign“ in der Compiler-Direktive enthält, müssen Sie sich nicht um den Speicher des Outlets kümmern. Ob Sie ein assign-Outlet verwenden hängt aber davon ab, ob das Objekt ohne ein retain-Outlet keinen Eigentümer mehr hätte. Verwenden Sie nach Möglichkeit immer Outlets mit retain-Properties.

In allen anderen Fällen („`retain`“, „`copy`“) müssen Sie sich um die Freigabe des Speichers kümmern, da es sich in dem Fall um eine starke Referenz handelt, welche den reference count um 1 erhöht.

```
@property (nonatomic, retain) IBOutlet UIView *chopperView;
@property (nonatomic, retain) UILabel *nameLabel;
```

Das Speichermanagement ist einfach, wenn Sie für Ihre Outlets immer `@property` und `@synthesize` Compiler-Direktiven schreiben. In der `-dealloc` Methode Ihrer Klasse schreiben Sie z.B.:

```
-(void)dealloc {
   [chopperView release], chopperView = nil;
   [nameLabel release], nameLabel = nil;

   [super dealloc]; //wichtig!
}
```

5.4 Starke Referenzen richtig freigeben

Wenn Sie sich ganz sicher sind, dass Sie ein Objekt nicht mehr benötigen, können Sie einige mysteriöse und sehr schwer identifizierbare Fehlerquellen vermeiden, wenn Sie sich die folgende

Vorgehensweise angewöhnen:

```
-(void)dealloc {
   [chopperView release], chopperView = nil;

   [super dealloc]; //wichtig!
}
```

Zunächst geben Sie ein Objekt wie üblich frei. Wenn es sonst niemanden gibt, der noch ein Interesse an dem Objekt haben könnte oder sollte, „zerstören“ Sie den Pointer auf das Objekt, in dem Sie `nil` zuweisen. `nil` steht für „kein Objekt“. Die Besonderheit an `nil` ist, dass das Objective-C Messaging ohne Fehler abbricht wenn `nil` der Empfänger einer Nachricht ist. Sie dürfen also beliebige Nachrichten an `nil` senden, ohne das es zu einem Programmcrash kommt.

Die Dereferenzierung eines `NULL`-Pointers bricht die Ausführung ab, während die Dereferenzierung eines Pointers, der auf ein nicht mehr existierendes Objekt verweist, zu mysteriösen, schwer nachvollziehbaren Fehlern führen kann.

Diese Vorgehensweise werden Sie leider nicht in allen iOS Codebeispielen antreffen. Wenn Sie sich damit unwohl fühlen, belassen Sie es bei `-release`. Es ist jedoch nie verkehrt, potenzielle Fehlerquellen zu reduzieren. Im Endeffekt sparen Sie dadurch viel Zeit und verbessern die Stabilität Ihrer Applikation.

Eine Alternative besteht darin, in -dealloc alle retain-Properties über deren Setter auf `nil` zu setzen:

```
-(void)dealloc {
   self.chopperView = nil;

   [super dealloc]; //wichtig!
}
```


Technisch gesehen bewirken Sie damit ziemlich genau das selbe wie in der vorher beschriebenen Variante, in der Sie ein Objekt erst durch `-release` freigeben und dann die Instanzvariable auf `nil` setzen. Obwohl diese Schreibweise sehr verlockend und wesentlich kürzer ist, gibt es Gründe die dagegen sprechen.

Wie Sie vielleicht bereits wissen, ermöglichen Properties das sogenannte Key-Value Observing, bei dem beliebige Objekte als Observer einer Property registrieren können, um bei Änderungen benachrichtigt zu werden. Damit Key-Value Observing funktioniert, senden die Setter entsprechende KVO-Notifications. Manchmal tun Setter auch noch andere Dinge. Alles Dinge, die Sie bei der Zerstörung des Objektes nicht unbedingt auslösen wollen.

Dagegen spricht, das Sie damit die Accessoren umgehen, was Sie im Normalfall nicht tun sollten. Allerdings ziehe ich in diesem Fall die Sicherheit und Code-Stabilität vor.

5.5 Collections

Collections beanspruchen grundsätzlich eine Eigentümerschaft an den Objekten, die ihnen übergeben werden. Sie erzeugen also starke Referenzen. Wird ein Objekt aus einer Collection entfernt, gibt die Collection die Eigentümerschaft an dem Objekt auf.

- Hinzufügen eines Objektes erhöht den retain count um 1.
- Entfernen eines Objektes verringert den retain count um 1.
- Wird eine `-release` Nachricht an eine Collection geschickt, betrifft dies nur die Collection. Erst wenn dadurch der retain count der Collection 0 wird, gibt die Collection ihre Eigentümerschaft an den in ihr enthaltenen Objekten auf. In diesem Fall sendet die Collection ihrerseits `-release` Nachrichten an die in ihr enthaltenen Objekte.

5.6 @“Stringkonstanten“

Sie müssen sich nicht um den Speicher dieser Strings kümmern. Es sei denn Sie referenzieren einen solchen String stark. Es handelt sich dabei um vollwertige `NSString` Instanzen.