Joe Conway
Aaron Hillegass

Big nerd ranch

iOS-Programmierung für iPhone und iPad

Der Big Nerd Ranch-Guide

ADDISON-WESLEY

ALWAYS LEARNING

PEARSON

iOS-Programmierung für iPhone und iPad

Aaron Hillegass, Joe Conway

iOS-Programmierung für iPhone und iPad

ADDISON-WESLEY

An imprint of Pearson

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

10 9 8 7 6 5 4 3 2 1

14 13 12

ISBN 978-3-8273-3015-4

ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachlektorat: Philipp Homann, Berlin
Korrektorat: Brigitte Hamerski, Willich
Covermotiv: Ellie Volckhausen
Covergestaltung: Marco Lindenbeck, mlindenbeck@webwo.de
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: mediaService, Siegen (www.mediaService.tv)
Druck und Verarbeitung: Kösel Druck, Krugzell (www.KoeselBuch.de)

Printed in Germany

3 Speicherverwaltung

Die Speicherverwaltung in Objective-C ist für Neulinge eine gewaltige Hürde. Anders als im Mac-Betriebssystem führt Objective-C unter iOS keine automatische Speicherbereinigung (garbage collection) durch. Sie müssen also selbst aufräumen.

3.1 Grundlagen der Speicherverwaltung

Wir setzen voraus, dass Sie C-Kenntnisse haben, sodass Ihnen die Begriffe »Zeiger«, »zuweisen« und »Zuweisung aufheben« vertraut sind. Für den Fall, dass Ihre Erinnerung ein wenig getrübt ist, geben wir zunächst einen Rückblick.

Ein iOS-Gerät hat eine begrenzte Menge Arbeitsspeicher (Random Access Memory, RAM), der sich wesentlich schneller beschreiben und auslesen lässt als eine Festplatte. Deshalb wird der gesamte Speicher, den eine Anwendung bei der Ausführung benutzt, aus dem RAM entnommen. Wenn ein Betriebssystem wie iOS Ihre Anwendung startet, reserviert es ihr eine Menge Speicher des nicht genutzten RAMs. Dieser reservierte Speicher wird nicht ganz zufällig als *Heap* bezeichnet. Er ist der Spielplatz Ihrer Anwendung. Sie kann damit machen, was sie will, ohne dass der Rest des Betriebssystems oder eine andere Anwendung gestört wird.

Wenn Ihre Anwendung eine Instanz einer Klasse anlegt, geht sie zu dem großen Speicherhaufen, den sie bekommen hat, und entnimmt ihm eine kleine Schaufel voll. Während der Ausführung erstellen Sie Objekte und nutzen einen immer größeren Teil des Heaps. Die meisten Objekte werden nicht dauerhaft benötigt, und der Speicher, den sie belegt hatten, sollte an den Heap zurückgegeben werden. Dann kann er für ein Objekt verwendet werden, das Sie später anlegen.

Diese dynamische Nutzung, Rückgabe und Wiederverwendung von Speicher setzt eine ordentliche Verwaltung voraus. Geschieht dies nicht, treten zwei wesentliche Probleme auf:

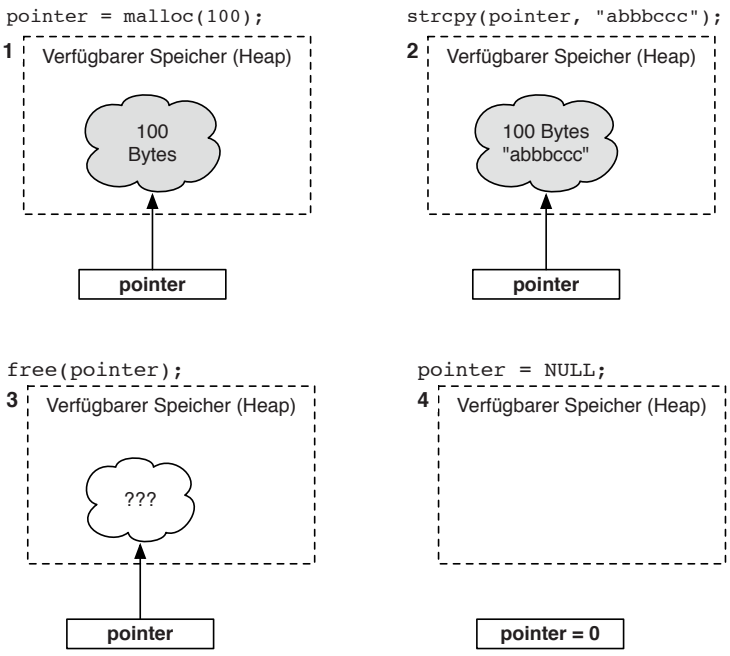
Vorzeitige Aufhebung der Zuweisung	Ein Speicherblock wird an den Heap zurückgegeben, obwohl ein Teil des Programms noch nicht mit der Nutzung fertig ist
Speicherleck	Ein Speicherblock wird von keinem Teil des Programms mehr benötigt, aber nicht freigegeben, um anderweitig verwendet zu werden

3.1.1 Speicherverwaltung in C

In der Programmiersprache C bitten Sie den Heap explizit um eine bestimmte Anzahl Bytes. Das wird als *Zuweisung* bezeichnet und ist das erste Stadium im Lebenslauf des Heaps, den Sie in Abbildung 3.1 sehen. Für die Speicherzuweisung verwenden Sie eine Funktion wie `malloc`. Wollen Sie 100 Bytes vom Heap haben, gehen Sie etwa wie folgt vor:

```
void function(void)
{
   char *buffer = malloc(100);
}
```

Sie haben dann 100 Bytes, mit denen Sie eine Aufgabe wie das Schreiben eines Strings und seine anschließende Ausgabe (die das Lesen der Bytes erfordert) durchführen können. Die Adresse des ersten der 100 Bytes wird im Zeiger `buffer` abgelegt. Mithilfe dieses Zeigers können Sie auf die 100 Bytes zugreifen (siehe Abbildung 3.1).



▶ *Abbildung 3.1: Lebenslauf der Heap-Zuweisung*

Benötigen Sie diese Bytes nicht mehr, geben Sie sie mithilfe der Funktion `free` an den Heap zurück, was als *Aufheben der Zuweisung* bezeichnet wird.

```
void function(void)
{
   char *buffer = malloc(100);
   // Mach etwas mit dem Puffer
   free(buffer);
}
```

Durch den Aufruf von `free` werden die 100 Bytes (beginnend mit der in `buffer` gespeicherten Adresse) an den Heap zurückgegeben. Wird eine weitere `malloc`-Funktion ausgeführt, steht ihr jedes dieser 100 Bytes für die Wiederverwendung zur Verfügung. Die Bytes können in kleinere Blöcke aufgeteilt oder Teil einer größeren Zuweisung werden. Da Sie nicht wissen, was aus diesen Bytes wird, wenn sie auf den Heap zurückkehren, ist es gefährlich, weiterhin mithilfe des Zeigers `buffer` auf sie zuzugreifen.

3.1.2 Speicherverwaltung mit Objekten

Obwohl ein Objekt auf der elementarsten Ebene auch eine bestimmte Anzahl von Bytes ist, die aus dem Heap zugewiesen wurden, rufen Sie `malloc` oder `free` niemals explizit für Objekte auf.

Jede Klasse weiß, wie viele Bytes Speicher sie für eine Instanz zuweisen muss. Wenn Sie mithilfe der Nachricht `alloc` eine Instanz einer Klasse anlegen, wird die korrekte Anzahl Bytes vom Heap zugewiesen. Wie bei `malloc` erhalten Sie einen Zeiger auf den Speicherblock. In Objective-C denken wir jedoch nicht in rohem Speicher, sondern in Objekten. Zwar zeigt unser Zeiger trotzdem auf eine Stelle im Arbeitsspeicher, aber wir brauchen die Einzelheiten dazu nicht zu kennen, sondern wissen einfach, dass wir ein Objekt haben.

Sobald Sie Speicher zuweisen, brauchen Sie natürlich eine Rückgabemöglichkeit. Jedes Objekt implementiert die Methode `dealloc`. Sie brauchen jedoch niemals eine `dealloc`-Nachricht an ein Objekt zu senden, weil das Objekt selbst dafür zuständig ist, `dealloc` an sich zu senden. Das wirft folgende Frage auf: Woher weiß ein Objekt, wann es gefahrlos und angebracht ist, sich zu zerstören, wenn es selbst dafür zuständig ist? Hier kommt die Referenzzählung ins Spiel.

3.2 Referenzzählung

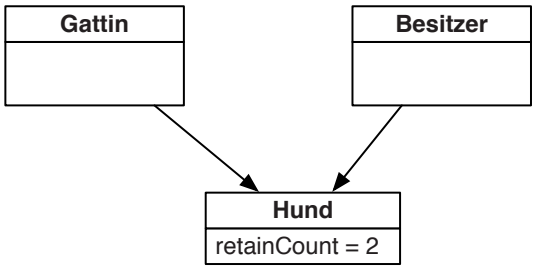
Im Framework Cocoa Touch hat Apple die *manuelle Referenzzählung* eingeführt, um den Speicher zu verwalten und die vorzeitige Aufhebung der Zuweisung sowie Speicherlecks zu vermeiden.

Um die Referenzzählung zu verstehen, stellen Sie sich einen kleinen Hund vor. Wenn er geboren wird, hat er einen Besitzer. Später heiratet dieser, und seine Gattin wird ebenfalls zum Besitzer des Hundes. Der Hund ist am Leben, weil die beiden ihn füttern. Später gibt das Paar den Hund weg. Der neue Besitzer beschließt, dass er den Hund nicht mag, und zeigt ihm dies, indem er ihn aus dem Haus wirft. Da er keinen Besitzer mehr hat, läuft der Hund weg und endet nach einer Reihe unglücklicher Ereignisse im Hundehimmel.

Wie lautet die Moral der Geschichte? Solange der Hund einen Besitzer hatte, der für ihn sorgte, war alles gut. Als er keinen Besitzer mehr hatte, lief er weg und hörte auf zu existieren. Genauso funktioniert die Referenzzählung. Wenn ein Objekt angelegt wird, hat es einen Besitzer. Im Lauf seiner Existenz kann es verschiedene Besitzer haben, sogar mehrere gleichzeitig. Hat es null Besitzer, dann hebt es die Zuweisung selbst auf und geht in den Instanzenhimmel.

3.2.1 Beibehaltungszähler verwenden

Ein Objekt weiß nie, *wem* es gehört. Es weiß nur, *wie viele* Besitzer es gerade hat. Diese Anzahl hält es in einem *Beibehaltungszähler* (*retain count*) fest (siehe Abbildung 3.2).



▶ *Abbildung 3.2: Beibehaltungszähler für einen Hund*

Wenn ein Objekt angelegt wird – und deshalb einen Besitzer hat –, wird sein Beibehaltungszähler auf 1 gesetzt. Gewinnt es einen Besitzer dazu, bekommt es die Nachricht `retain`, und der Zähler wird hoch gesetzt. Verliert ein Objekt einen Besitzer, so bekommt es die Nachricht `release`, woraufhin der Beibehaltungszähler herabgesetzt wird. Erreicht der Zähler 0, sendet das Objekt selbst die Nachricht `dealloc`, was den gesamten Speicher, den es belegt hat, an den Heap zurückgibt.

Stellen Sie sich vor, wie Sie den Code zur Implementierung dieses Systems schreiben würden:

```
- (id)retain
{
   retainCount++;
   return self;
}
- (void)release
{
   retainCount--;
   if (retainCount == 0)
      [self dealloc];
}
```

Doch wie funktionieren Beibehaltungszähler zwischen Objekten? Stellen Sie sich dazu einen Einkaufszettel vor. Sie haben ihn angelegt, sind also der Besitzer. Später geben Sie Ihrem Freund den Einkaufszettel, damit er den Einkauf erledigt. Sie brauchen den Zettel nicht aufzuheben und geben ihn frei. Ihr Freund ist intelligent und behält den Zettel, nachdem er ihn bekommen hat. Deshalb existiert er weiter, solange Ihr Freund ihn benötigt, und nun ist er der einzige Besitzer des Zettels.

So sieht Ihr Code aus:

```
- (void)createAndGiveAwayTheGroceryList
{
   // Legt einen Einkaufszettel an
   GroceryList *g = [[GroceryList alloc] init];

   // (Der Beibehaltungszähler von g ist 1)

   // Gibt ihn dem Freund, der ihn behält
    [smartFriend takeGroceryList:g];

   // (Der Beibehaltungszähler von g ist 2)

   // Gibt den Besitz auf
   [g release];

   // (Der Beibehaltungszähler von g ist 1)
   // Es kommt hier aber nicht mehr darauf an, weil
   // die Zuständigkeit dieser Methode vorbei ist.
}
```

So sieht der Code Ihres Freundes aus:

```
- (void)takeGroceryList:(GroceryList *)x
{
   // Übernimmt den Besitz
   [x retain];

   // Behält einen Zeiger auf ein Objekt bei
   myList = x;
}
```

Bei Beibehaltungszählern kann es trotzdem auf die beiden klassischen Arten zu Fehlern kommen: durch Speicherlecks und durch vorzeitige Aufhebung von Zuweisungen. Bleiben wir beim Beispiel mit dem Einkaufszettel und nehmen wir an, Sie schreiben einen Einkaufszettel und geben ihn Ihrem Freund. Er behält ihn, aber Sie geben ihn nicht frei. Ihr Freund schließt den Einkauf ab und gibt den Zettel frei. Inzwischen haben Sie vergessen, wo der Zettel ist, und weil Sie ihn nie freigegeben haben, ist sein Beibehaltungszähler größer 0 – und bleibt es. Im Augenblick weiß niemand, wo der Zettel ist, aber es gibt ihn noch. Das ist ein Leck.

Stellen Sie sich den Einkaufszettel als `NSArray` vor. In der Methode, in der Sie das Array erstellt haben, verfügen Sie über einen Zeiger darauf. Verlassen Sie den Gültigkeitsbereich der Methode, ohne das `NSArray`-Objekt freizugeben, verlieren Sie sowohl den Zeiger als auch die Fähigkeit, das Objekt später freizugeben. (Eine Freigabenachricht können Sie nur senden, wenn Sie wissen, *wohin* sie gehen soll.) Auch wenn jeder andere Besitzer das `NSArray`-Objekt freigibt, wird die Zuweisung niemals aufgehoben, und die Anwendung kann den Speicherblock nicht für etwas anderes verwenden.

Betrachten Sie die andere Art und Weise, in der dieser Vorgang falsch ablaufen kann – die vorzeitige Aufhebung der Zuweisung. Sie schreiben einen Einkaufszettel und geben ihn einem Freund, der ihn nicht aufhebt. Wenn Sie den Zettel freigeben, wird die Zuweisung aufgehoben, weil Sie der einzige Besitzer waren. Versucht Ihr Freund später, den Zettel zu benutzen, kann er ihn nicht finden, weil er nicht mehr existiert.

Diese Situation ist schlimmer, als sie klingt. Ihr Freund ist nicht nur nicht in der Lage, den Einkauf zu erledigen, sondern die vorzeitige Aufhebung der Zuweisung hat auch Folgen für die Anwendung. Wenn ein Objekt versucht, auf ein anderes Objekt zuzugreifen, das nicht existiert, greift die Anwendung auf unzugewiesenen Speicher zu, beginnt fehlzuschlagen und stürzt schließlich ab. Behält ein Objekt dagegen die Objekte bei, die es benötigt, ist deren Existenz garantiert, sodass diese Art von Katastrophe vermieden wird.

Bei der Referenzzählung geht es um Zuständigkeit: Wenn etwas ein Objekt anlegt, ist es Besitzer des Objekts. Dasselbe gilt für die Beibehaltung eines vorhandenen Objekts. Das Freigeben eines Objekts beendet den Besitz. Wenn etwas den Besitz an einem Objekt übernimmt, ist es dafür verantwortlich, seinen Besitz zu beenden, wenn es keine Nachrichten mehr an das Objekt senden kann, also wenn es keinen Zeiger auf das Objekt mehr hat.

Sehen wir uns diese theoretischen Konzepte am Beispiel des Programms `RandomPossessions`, das Sie im vorigen Kapitel geschrieben haben, konkret an. Öffnen Sie `RandomPossessions.xcodeproj` und dann `main.m` im Editorbereich. In der Funktion `main` haben Sie eine Instanz von `NSMutableArray` mit dem Namen `items` angelegt. Über diese Instanz wissen Sie zwei Dinge: Die Funktion `main` ist ihr Besitzer, und ihr Beibehaltungszähler steht auf 1. Es liegt in der Zuständigkeit von `main` als Besitzer, dieser Instanz die Nachricht `release` zu senden, wenn sie sie nicht mehr braucht. Zum letzten Mal verweisen Sie in dieser Funktion auf `items`, wenn Sie alle Einträge darin ausgeben, und können Sie deshalb anschließend freigeben:

```
for (Possession *item in items) {
   NSLog(@"%@", item);
}
[items release];
items = nil;
```

Wenn die Nachricht `release` gesendet wird, setzt das Objekt, auf das `items` zeigt, seinen Beibehaltungszähler herab. In diesem Fall wird die Zuweisung des Objekts aufgehoben, weil `main` sein einziger Besitzer war. Hätte ein anderes Objekt `items` beibehalten, wäre die Zuweisung nicht aufgehoben worden.

3.2.2 Die automatische Freigabe nutzen

Sie haben `items` in `main` angelegt, verwenden es dort und geben es dort frei. Was geschieht aber, wenn Sie ein Objekt erstellen wollen, um es weiterzugeben, anstatt als sein Besitzer zu fungieren und es selbst zu verwenden?

Dies kommt häufig bei Komfortmethoden vor – Klassenmethoden, die Instanzen der Klasse zurückgeben. In der Klasse `Possession` haben Sie die Komfortmethode `randomPossession` implementiert, die eine Instanz von `Possession` mit zufälligen Parametern zurückgibt. Die Klasse `Possession` ist Besitzer dieser Instanz, weil sie innerhalb einer Klassenmethode von `Possesssion` erstellt wurde, und der Beibehaltungszähler der Instanz steht auf 1.

Die Klasse `Possession` selbst hat jedoch keine Verwendung für diese Instanz. `randomPossession` wird von `main` aufgerufen und gibt die neu erstellte Instanz von `Possession` dorthin zurück.

Sollten Sie das `Possession`-Objekt in `main` freigeben?

```
for(int i = 0; i < 10; i++)
{
   Possession *p = [Possession randomPossession];
   [items addObject:p];

   // Tun Sie das nicht!
   [p release];
}
```

Das ist eine ganz schlechte Idee. Das Prinzip der Zuständigkeit, auf das sich die ganze Referenzzählung stützt, verlangt u.a., keine Objekte freizugeben, die einem nicht gehören. Das kommt dem Absagen der Party eines Freundes gleich. Oder dem Abgeben des Nachbarhundes im Tierheim. Gehört Ihnen ein Objekt nicht, dann sollten Sie es nicht freigeben, und diese Instanz von `Possession` gehört `main` nicht, denn `main` hat sie weder zugewiesen noch beibehalten, sondern verfügt dank der Komfortmethode `randomPossession` nur über einen Zeiger auf sie.

Die `Possession`-Instanz freizugeben ist Sache der Klasse `Possession`, und es muss in `randomPossession` erfolgen, bevor der Zeiger auf die Instanz verlorengeht, wenn die Methode ihren Gültigkeitsbereich verlässt. Aber an welcher Stelle in `randomPossession` können Sie die neue `Possession`-Instanz gefahrlos freigeben?

```
+ (id)randomPossession
{
   ... Hier werden zufällige Variablen angelegt ...
   Possession *newPossession = [[self alloc]
                        initWithPossessionName:randomName
                                valueInDollars:randomValue
                                  serialNumber:randomSerialNumber];

   // Geben wir newPossession hier frei,
   // wird die Zuweisung des Objekts aufgehoben,
   // bevor es zurückgegeben wird.

   return newPossession;

   // Geben wir newPossession hier frei, wird dieser Code nie
```

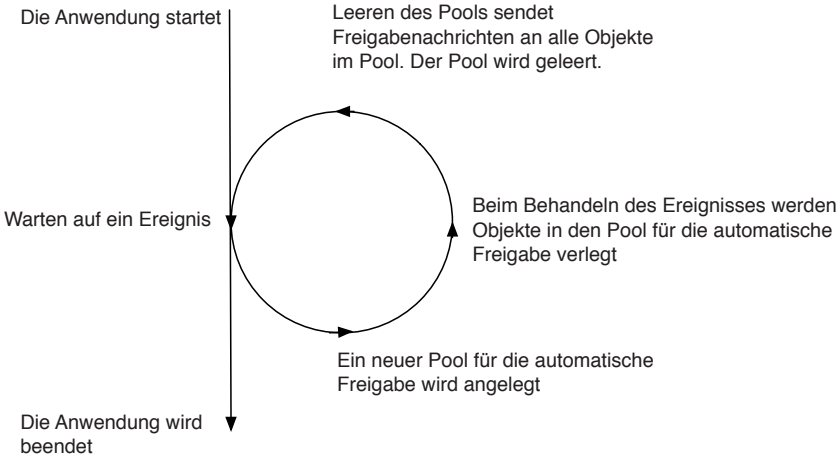
```
    // ausgeführt.
}
```

Was können Sie tun?

Sie brauchen eine Möglichkeit zu sagen: »Gib dies Objekt noch nicht frei, aber lass mich nicht mehr sein Besitzer sein.« Glücklicherweise können Sie ein Objekt für die spätere Freigabe markieren, indem Sie ihm die Nachricht `autorelease` senden. Dann wird es nicht sofort freigegeben, sondern in eine Instanz von `NSAutoreleasePool` aufgenommen. Dieser Pool für die automatische Freigabe verfolgt alle Objekte nach, die eine `autorelease`-Nachricht erhalten haben. Er wird in bestimmten Abständen geleert, indem er den in ihm enthaltenen Objekten eine `release`-Nachricht sendet und sie dann löscht.

Für ein Objekt, das nach der Erstellung für die automatische Freigabe markiert wird, gibt es zwei mögliche Schicksale: Entweder setzt es seinen letzten Gang bis zur Aufhebung der Zuweisung fort, oder ein anderes Objekt kann es beibehalten. Im zweiten Fall steht sein Beibehaltungszähler zunächst auf 2. (Das beibehaltende Objekt ist sein Besitzer, und der Pool für die automatische Freigabe hat noch keine `release`-Nachricht gesendet.) Irgendwann später gibt der Pool es frei, sodass der Beibehaltungszähler auf 1 zurückgesetzt wird.

Manchmal sorgt die Vorstellung, dass Objekte zu irgendeinem späteren Zeitpunkt freigegeben werden, bei Entwicklern für Verwirrung. Während eine iOS-Anwendung läuft, wird eine Ausführungsschleife ständig wiederholt. Sie prüft auf Ereignisse wie eine Berührung oder das Auslösen eines Timers. Sobald ein Ereignis eintritt, bricht sie die Ausführungsschleife ab und verarbeitet das Ereignis, indem sie die Methoden aufruft, die Sie in Ihre Klassen geschrieben haben. Ist die Ausführung Ihres Codes abgeschlossen, kehrt die Anwendung zur Schleife zurück. Am Ende der Schleife wird die Nachricht `release` an alle Objekte mit automatischer Freigabe gesendet (siehe Abbildung 3.3). Während Sie eine Methode ausführen, die ihrerseits andere Methoden aufrufen kann, können Sie also sicher davon ausgehen, dass ein Objekt mit automatischer Freigabe nicht freigegeben wird.



▶ *Abbildung 3.3: Den Pool für die automatische Freigabe leeren*

Der Rückgabewert für `autorelease` ist die Instanz, an die die Nachricht geht. Sie können `autorelease`-Nachrichten also verschachteln.

```
// Da die automatische Freigabe das freizugebende Objekt zurückgibt,
// können wir Folgendes tun:
NSObject *x = [[[NSObject alloc] init] autorelease];
```

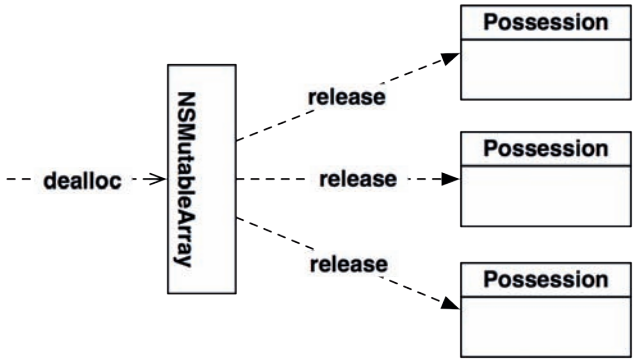
Setzen Sie die neu erstellte Instanz am Ende von `randomPossession` in `Possession.m` auf automatische Freigabe, sodass der Empfänger des Objekts die Wahl hat, ob er es behält oder einfach zerstören lässt.

```
    Possession *newPossession =
        [[self alloc] initWithPossessionName:randomName
                                valueInDollars:randomValue
                                  serialNumber:randomSerialNumber];

    return [newPossession autorelease];
}
```

Wenn die Funktion `main` die Klasse `Possession` um ein zufälliges Besitztum bittet, gibt die Klasse nun in `main.m` eine Instanz von `Possession` mit automatischer Freigabe zurück. Zu diesem Zeitpunkt hat die Instanz keinen Besitzer. Wenn das `Possession`-Objekt in das Array `items` aufgenommen wird, behält das Array es bei, und sein Beibehaltungszähler steht auf 1.

```
for(int i = 0; i < 10; i++)
{
    // Ruft eine neue Instanz von Possession ab. Sie gehört niemandem,
    // weil sie auf automatische Freigabe gesetzt wurde.
    Possession *p = [Possession randomPossession];

    // Fügt p in das Array items ein, das das Possession-Objekt beibehält
    [items addObject:p];
}
```

Wann gibt `items` das `Possession`-Objekt frei? Wenn die Zuweisung eines `NSMutableArray` aufgehoben wird, werden die darin enthaltenen Objekte freigegeben. Gibt `main` (der einzige Besitzer von `items`) `items` frei, dann gibt das `NASMutableArray`, auf das `items` zeigt, also all seine `Possession`-Instanzen frei (siehe Abbildung 3.4). Wir haben uns davon überzeugt, dass diese `Possesssion`-Instanzen nur `items` als Besitzer hatten, sodass ihre Zuweisung aufgehoben wird.



▶ *Abbildung 3.4: Die Zuweisung eines NSMutableArray aufheben*

Die folgenden drei Regeln der Speicherverwaltung sollten Sie sich merken, wenn Sie mit Instanzen von `NSMutableArray` arbeiten:

- **Wenn ein Objekt in ein `NSMutableArray` aufgenommen wird, erhält es die Nachricht `retain`. Das Array wird zum Besitzer des Objekts und hat einen Zeiger darauf.**
- **Wenn ein Objekt aus einem `NSMutableArray` gelöscht wird, erhält es die Nachricht `release`. Das Array verliert den Besitz an diesem Objekt und hat keinen Zeiger mehr darauf.**
- **Wenn die Zuweisung eines `NSMutableArray` aufgehoben wird, sendet es all seinen Einträgen die Nachricht `release`.**

Wenden wir uns nun einer anderen Stelle in `randomPossessions` zu, an der Sie `autorelease` benutzen sollten. In `Possession.m` überschreiben Sie die Methode `description` der übergeordneten Klasse von `Possession`. Diese Methode erstellt eine Instanz von `NSString` und gibt sie zurück. Ändern Sie die Methode so, dass sie einen String mit automatischer Freigabe zurückgibt.

```
- (NSString *)description
{
    NSString *descriptionString =
        [[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, Recorded on
%@",
                                possessionName,
                                serialNumber,
                                valueInDollars,
                                dateCreated];
    return [descriptionString autorelease];
}
```

Durch Verwendung einer Komfortmethode können Sie dies noch vereinfachen. Wie viele andere Klassen im iOS SDK enthält `NSString` Komfortmethoden, die Objekte mit automatischer Freigabe zurückgeben, wie `randomPossession` es jetzt tut. Ändern Sie die Methode `description` so, dass sie die Komfortmethode `stringWithFormat:` verwendet. Dadurch wird sichergestellt, dass die von `description` angelegte und zurückgegebene `NSString`-Instanz mit automatischer Freigabe versehen ist.

```
- (NSString *)description
{
   return [NSString stringWithFormat:@"%@ (%@): Worth $%d, Recorded on %@",
                        possessionName,
                        serialNumber,
                        valueInDollars,
                        dateCreated];
}
```

3.2.3 Zugriffsmethoden und Speicherverwaltung

Bis jetzt ging es in unseren Beispielen um den Besitz durch Erstellen. Wollen Sie Besitzer eines Objekts werden, das Sie nicht angelegt haben, müssen Sie es beibehalten. Hat ein Objekt zum Beispiel Instanzvariablen, die auf andere Objekte zeigen, sollte es diese beibehalten. Ein Objekt, auf das eine Instanzvariable zeigt, lässt sich in der Set-Methode für die betreffende Variable beibehalten.

Sehen wir uns die Instanzvariablen für das `Possession`-Objekt an. Jede Instanz von `Possession` hat drei Instanzvariablen, die Zeiger auf andere Objekte sind (`possessionName`, `serialNumber` und `dateCreated`). Im Augenblick weisen die Set-Methoden in `Possession` lediglich den eingehenden Wert der Instanzvariable zu:

```
- (void)setPossessionName:(NSString *)str
{
   possessionName = str;
}
```

Das reicht nicht. Wenn wir einem `Possession`-Objekt ein `NSString`-Objekt für die Variable `possessionName` geben und das `Possession`-Objekt es nicht beibehält, wird es zerstört, wenn wir den String freigeben. Das `Possession`-Objekt sendet schließlich Nachrichten oder gibt seine Variable `possessionName` weiter. Es ist katastrophal (denn es bringt die Anwendung zum Absturz), wenn das Objekt, auf das die Variable zeigt, nicht existiert.

Deshalb sollte ein Objekt in einer Set-Methode die Objekte beibehalten, auf die seine Instanzvariablen zeigen, um sicherzustellen, dass die Objekte weiterbestehen. Öffnen Sie `Possession.m` im Editorbereich und ändern Sie die Methode `setPossessionName:` so, dass das `Possession`-Objekt den ihm übergebenen String beibehält:

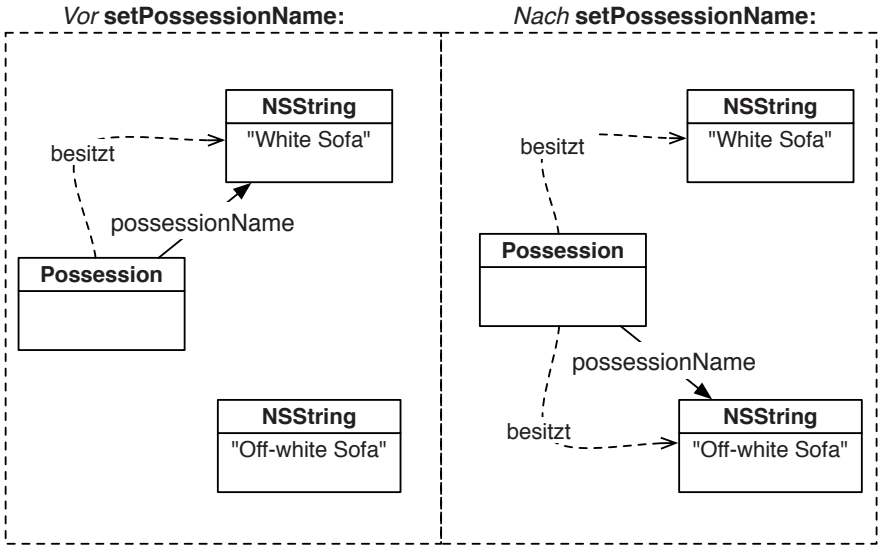
```
- (void)setPossessionName:(NSString *)str
{
   [str retain];
   possessionName = str;
}
```

Das `Possession`-Objekt setzt den Beibehaltungszähler des ihm übergebenen Strings hoch, und trotz allem, was an anderer Stelle im Code mit dem String geschieht, existiert er fort, bis das `Possession`-Objekt ihn freigibt.

Sehen wir uns nun an, was geschieht, wenn Sie die Variable mithilfe der Set-Methode ändern. Stellen wir uns zum Beispiel vor, wir haben ein Besitztum »White Sofa« genannt, und ein schlauer Freund weist uns darauf hin, dass es eigentlich cremefarben (»off-white«) ist:

```
Possession *p = [[Possession alloc] init];
[p setPossessionName:@"White Sofa"];

// Halt, es ist gar nicht weiß ...
[p setPossessionName:@"Off-white Sofa"];
```

In diesem Code behält `p` den String »White Sofa« bei und lässt die Variable `possessionName` darauf zeigen. Dann behält `p` den String »Off-white Sofa« bei und lässt die Variable `possessionName` jetzt auf den neuen String zeigen. Das ist ein Leck: Das `Possession`-Objekt hat seinen Zeiger auf den String »White Sofa« verloren, jedoch nie den Besitz daran freigegeben (siehe Abbildung 3.5).



▶ *Abbildung 3.5: Eine Set-Nachricht mehrmals senden*

Deshalb behalten Sie in einer Set-Methode das neue Objekt bei, geben das Objekt frei, das Sie gerade haben, und nehmen dann die Zuweisung vor. Fügen Sie die folgende Codezeile in `Possession.m` in diese Set-Methode ein.

```
- (void)setPossessionName:(NSString *)str
{
   [str retain];
   [possessionName release];
   possessionName = str;
}
```

Sie müssen das neue Objekt beibehalten, bevor Sie das aktuelle freigeben. `possessionName` und `str` zeigen häufiger auf dasselbe Objekt, als Sie sich vorstellen. Vertauschen Sie die Beibehaltungs- und die Freigabeanweisung, geben Sie das Objekt frei, das Sie als `possessionName` beibehalten wollten. O Schreck!

Schreiben Sie jetzt den entsprechenden Code für die Methode `setSerialNumber:` in `Possession.m`.

```
- (void)setSerialNumber:(NSString *)str
{
    [str retain];
    [serialNumber release];
    serialNumber = str;
}
```

Was geschieht in diesen Set-Methoden, wenn das eingehende Argument oder die aktuelle Instanzvariable auf `nil` zeigt? Übergeben Sie in `setPossessionName: nil` als Argument, gibt das `Possession`-Objekt seinen aktuellen `possessionName` frei und setzt seinen `possessionName` auf `nil`. Als Ergebnis davon hat das `Possession`-Objekt keinen Namen. Senden Sie `setPossessionName:` an ein `Possession`-Objekt, das keinen `possessionName` hat, dann senden Sie die `release`-Nachricht an `nil`, was nichts bewirkt.

Beachten Sie, dass wir nur Set-Methoden geändert haben. Get-Methoden bedürfen keiner weiteren Speicherverwaltung. Das Objekt, das die Get-Nachricht sendet, muss jedoch möglicherweise das beibehalten, was zurückgegeben wird.

Was ist mit den beiden anderen Instanzvariablen? `dateCreated` hat keine Set-Methode, sondern wird angelegt und bekommt ihren Wert im vorgesehenen Initialisierer. Deshalb ist die Instanz von `Possession` bereits ihr Besitzer, und ihre Existenz ist garantiert. Die Instanzvariable `valueInDollars` benötigt keine Speicherverwaltung, weil sie kein Objekt, sondern ein Primitivelement ist.

3.2.4 Die Methode dealloc implementieren

Im vorigen Abschnitt haben Sie das Objekt freigegeben, auf das `possessionName` zeigte, als Sie den Namen eines `Possession`-Objekts geändert haben. Genauso wichtig ist es, die Objekte freizugeben, auf die die Instanzvariablen eines `Possession`-Objekts zeigen, wenn seine Zuweisung aufgehoben wird.

Wenn der Beibehaltungszähler einer `Possession`-Instanz auf 0 geht, sendet er selbst die Nachricht `dealloc`. Bei der Zerstörung der `Possession`-Instanz werden auch ihre Instanzvariablen zerstört, die Zeiger auf andere Objekte sind (die Objekte, auf die sie zeigen, aber nicht). Daher müssen Sie sich fragen, ob das `Possession`-Objekt Besitzer der Objekte ist, und sie freigeben, wenn dies der Fall ist, bevor die Zeiger zerstört werden.

Die Objekte, auf die possessionName und serialNumber zeigen, gehören Ihnen, weil Sie sie in ihren Set-Methoden beibehalten haben. Das gilt auch für dateCreated, weil Sie es im vorgesehenen Initialisierer für Possession zugewiesen haben.

Nachdem Sie festgestellt haben, dass Sie der Besitzer sind, müssen Sie diese Objekte freigeben, bevor Sie die Zeiger darauf verlieren. Sie können dies am Anfang der Methode dealloc von Possession erledigen. Überschreiben Sie dealloc in Possession.m, um die Instanzvariablen freizugeben, die dem Possession-Objekt gehören.

```
- (void)dealloc
{
   [possessionName release];
   [serialNumber release];
   [dateCreated release];
   [super dealloc];
}
```

Rufen Sie am Ende der Methode grundsätzlich die dealloc-Implementierung der übergeordneten Klasse auf. Wenn die Zuweisung eines Objekts aufgehoben wird, sollte es zunächst seine Instanzvariablen freigeben. Da Sie die Implementierung der übergeordneten Klasse aufrufen, geht es dann in der Klassenhierarchie nach oben und gibt alle Instanzvariablen seiner übergeordneten Klasse frei. Am Ende gibt die Implementierung von dealloc in NSObject den Speicher des Objekts an den Heap zurück.

Warum senden Sie keine dealloc-, sondern release-Nachrichten an Instanzvariablen? Ein Objekt sollte niemals eine dealloc-Nachricht an ein anderes Objekt schicken. Verwenden Sie immer release und lassen Sie das Objekt seinen Beibehaltungszähler prüfen und entscheiden, ob es sich selbst eine dealloc-Nachricht sendet.

3.2.5 Zugriffsmethoden mit Eigenschaften vereinfachen

Nachdem Sie Ihre Set-Methoden nun mit einer Speicherverwaltung ausgestattet haben, sehen wir uns eine Abkürzung zum Erstellen von Zugriffsmethoden (für Get- und Set-Methoden) an, nämlich Eigenschaften. Eine *Eigenschaft* deklariert Zugriffsmethoden in einer Headerdatei. Ersetzen Sie die Deklarationen der Zugriffsmethoden in Possession.h durch Eigenschaften.

```
@interface Possession : NSObject
{
   NSString *possessionName;
   NSString *serialNumber;
   int valueInDollars;
   NSDate *dateCreated;
}
+ (id)randomPossession;

- (id)initWithPossessionName:(NSString *)name
              valueInDollars:(int)value
```

```
                serialNumber:(NSString *)sNumber;

- (id)initWithPossessionName:(NSString *)name;

@property NSString *possessionName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;

@end
```

Beachten Sie, dass Eigenschaften nicht zusammen mit den Instanzvariablen in den geschweiften Klammern, sondern im Methodenbereich deklariert werden. Außerdem stehen sie vereinbarungsgemäß hinter Klassenmethoden und Initialisierern, aber vor anderen Instanzmethoden.

Eigenschaften ersetzen die Deklaration der Zugriffsmethoden, was einige Zeilen in der Headerdatei spart. Aber das ist nicht alles. Mit Eigenschaften können Sie auch die Implementierungen der Zugriffsmethoden automatisch erstellen. Dazu *synthetisieren* Sie die Eigenschaft in der Implementierungsdatei. Aber bevor wir dazu kommen, müssen wir uns mit *Eigenschaftsattributen* befassen.

Jede Eigenschaft hat eine Reihe von Attributen, um die Zugriffsmethoden maßzuschneidern, die sie erzeugen kann. Diese Attribute sind in der ihrer Deklaration aufgeführt. Es gibt drei Kategorien:

Atomizität	Für dieses Attribut verwenden wir immer `nonatomic`. Es gibt selten einen Grund, den Standardwert `atomic` zu verwenden, aber eine genauere Erklärung würde den thematischen Rahmen dieses Buches sprengen.
Beschreibbarkeit	Standardmäßig sind Eigenschaften auf `readwrite` gesetzt. Solche Eigenschaften erstellen eine Set- und eine Get-Methode. Die andere Option, `readonly`, legt nur eine Get-Methode an.
Speicherverwaltung	Diese Attributkategorie betrifft nur Set-Methoden. Der Standardwert lautet `assign`. In diesem Fall weist die Set-Methode einer Eigenschaft nur das eingehende Argument seiner Instanzvariable zu. Die anderen Optionen sind `retain` und `copy`, bei denen das eingehende Argument entweder beibehalten oder kopiert und anschließend der Instanzvariable zugewiesen wird. (Über `copy` erfahren Sie in Kürze mehr.)

Versehen Sie die Eigenschaftsdeklarationen in `Possession.h` mit Attributen, die den aktuellen Implementierungen der Zugriffsmethoden entsprechen.

```
// Für diese Eigenschaft werden eine Get- und eine Set-Methode erstellt,
// die das eingehende Objekt beibehält und das alte freigibt.
@property (nonatomic, retain) NSString *possessionName;
```

```
// Wie bei der vorstehenden Eigenschaft
@property (nonatomic, retain) NSString *serialNumber;

// Für diese Eigenschaft werden eine Get- und eine Set-Methode erstellt,
// die den eingehenden Wert einfach der ivar-Variable valueInDollars
// zuweist.
@property (nonatomic) int valueInDollars;

// Für diese Eigenschaft wird nur eine Zugriffsmethode erstellt, nämlich
// eine Get-Methode.
@property (nonatomic, readonly) NSDate *dateCreated;
```

Jetzt können wir die Eigenschaften in der Implementierungsdatei synthetisieren. Löschen Sie die Implementierungen der Zugriffsmethoden komplett aus `Possession.h` und synthetisieren Sie stattdessen die Eigenschaften.

```
@implementation Possession
@synthesize possessionName, serialNumber, valueInDollars, dateCreated;
```

Erstellen Sie einen Build der Anwendung und führen Sie sie aus. Alles sollte genauso funktionieren wie vorher.

Gehen wir noch einmal durch, was sich geändert hat. Vorher haben Sie sämtliche Zugriffsmethoden explizit deklariert und implementiert. Jetzt haben Sie die Deklaration der Zugriffsmethoden durch Eigenschaftsdeklarationen und die Implementierung der Zugriffsmethoden durch eine `@synthesize`-Anweisung ersetzt. Das `Possession`-Objekt verhält sich bei erheblich weniger Schreibarbeit genauso wie vorher. Sobald Sie in der Programmierung Details angeben und dem System die Arbeit überlassen können, spart es Ihnen nicht nur Eingaben, sondern trägt auch zur Vermeidung von Tipp- und anderen Fehlern bei.

Warum haben wir Sie dann zunächst alle Zugriffsmethoden schreiben lassen, anstatt direkt zu den Eigenschaften zu gehen? Sie müssen erst wissen, was Eigenschaften eigentlich tun. Zu viele neue Entwickler verwenden Eigenschaften, ohne den zugrunde liegenden Code zu verstehen, und stolpern später darüber.

Es gibt noch einiges mehr über Eigenschaften zu sagen. Erstens müssen Sie nicht unbedingt eine Eigenschaft synthetisieren, sondern können sie auch in der Headerdatei deklarieren und die Zugriffsmethoden selbst implementieren. Das ist in Situationen sinnvoll, in denen Sie die Zugriffsmethoden anpassen wollen. Sie können auch eine Eigenschaft synthetisieren und eine der Zugriffsmethoden implementieren, womit Sie die Methode überschreiben, die Sie ersetzt haben, ohne ihren Partner zu beeinträchtigen.

Zweitens muss der Name der Eigenschaft nicht mit dem Namen der Instanzvariable identisch sein. In einer Synthetisierungsanweisung können Sie eine Eigenschaft auf eine Instanzvariable mit einem anderen Namen zeigen lassen.

```
// Dies ist nur ein Beispiel, geben Sie diesen Code nicht ein.
@interface Possession : NSObject
{
    ...
}
@property (nonatomic, assign) NSString *name;
@end

@implementation Possession

@synthesize name = possessionName;
// Dies entspricht:
// - (void)setName:(NSString *)str
// {
//    possessionName = str;
// }
// - (NSString *)name
// {
//    return possessionName;
// }
@end
```

Damit verknüpfen Sie die Eigenschaft `name` mit der Instanzvariable `possessionName`. Wenn Sie die Nachricht `name` an eine Instanz von `Possession` senden, wird deshalb der Wert von `possessionName` zurückgegeben.

Schließlich brauchen Sie noch nicht einmal eine Instanzvariable. Wenn Sie eine Eigenschaft synthetisieren, sucht der Compiler nach einer Instanzvariable desselben Namens und verwendet sie, wenn er eine findet. Findet er keine passende Instanzvariable, wird automatisch eine angelegt.

3.2.6 copy und mutableCopy

Gelegentlich müssen Sie ein Objekt kopieren, anstatt es beizubehalten. Wenn Sie die Nachricht `copy` an eine Instanz senden, wird eine brandneue Instanz angelegt, die dieselben Werte hat wie die ursprüngliche. Durch das Kopieren erhalten Sie ein neues Objekt mit dem Beibehaltungswert 1, während der Beibehaltungszähler des Ausgangsobjekts unverändert bleibt. Das Objekt, das die `copy`-Nachricht gesendet hat, ist Besitzer des neuen Objekts.

Normalerweise machen Sie eine Kopie eines Objekts, wenn es *veränderbar* ist. Beispielsweise ist ein `NSMutableArray`-Objekt ein veränderbares Array und `NSMutableString` eine veränderbare Unterklasse von `NSString`. Da `NSMutableString` ein `NSString`-Objekt ist, kann eine Instanz von `NSMutableString` als `possessionName` dienen.

Stellen Sie sich vor, was passiert, wenn ein NSMutableString-Objekt als possessionName eines Possession-Objekts festgelegt wird. Ein anderes Objekt, das einen Zeiger auf diesen String hat, kann ihn ändern, was dann auch den Namen des Possession-Objekts ändert. Das Possession-Objekt weiß aber nicht, dass dies geschehen ist.

```
NSMutableString *str = [[NSMutableString alloc]
                        initWithString:@"White Sofa"];

// Das ist in Ordnung, da NSMutableString eine untergeordnete Klasse von
// NSString und damit ein NSString-Objekt ist
[possession setPossessionName:str];

[str appendString:@" - Stained"];
// Der Name des Besitztums lautet jetzt "White Sofa - Stained"
```

Normalerweise ist dieses Verhalten nicht erwünscht. Instanzvariablen eines Objekts ohne Verwendung einer Zugriffsmethode zu ändern, gilt als schlechter Stil. Mithilfe von copy können Sie verhindern, dass Instanzvariablen hinter Ihrem Rücken geändert werden.

Wenn eine Klasse eine veränderbare Unterklasse hat, sollten Eigenschaften, die vom selben Typ wie die Klasse sind, im Allgemeinen nicht das Attribut retain, sondern copy tragen. Dann haben Sie eine Kopie des Objekts, die allein Ihnen gehört. Ändern Sie das Speicherverwaltungsattribut der beiden NSString-Eigenschaften in Possession.h auf copy.

```
@property (nonatomic, copy) NSString *possessionName;
@property (nonatomic, copy) NSString *serialNumber;
```

Die erzeugten Set-Methoden für diese Eigenschaften sehen jetzt wie folgt aus:

```
- (void)setPossessionName:(NSString *)str
{
   id t = [str copy];
   [possessionName release];
   possessionName = t;
}
```

Wenn Sie ein Objekt kopieren, ist die zurückgegebene Kopie unveränderlich. Kopieren Sie zum Beispiel ein NSMutableArray, ist das neue Objekt einfach ein NSArray. (Soll die Kopie verändert werden können, müssen Sie stattdessen die Nachricht mutableCopy senden.) Denken Sie daran, dass nicht alle Klassen veränderbare Unterklassen haben und nicht alle Objekte kopiert werden können.

Herzlichen Glückwunsch! Sie haben Beibehaltungszähler implementiert und die Speicherverwaltungsprobleme in RandomPossessions behoben. Ihre Anwendung verwaltet den Arbeitsspeicher jetzt meisterhaft!

Halten Sie diesen Code griffbereit, denn Sie werden ihn in späteren Kapiteln noch benötigen.

3.2.7 Regeln für den Beibehaltungszähler

Prägen Sie sich die folgenden Regeln über Beibehaltungszähler gut ein. In diesen Regeln benutzen wir das Wort »Sie« für »eine Instanz der Klasse, mit der Sie gerade arbeiten«. Das ist eine sinnvolle Form von Personifizierung: Sie stellen sich vor, Sie seien das Objekt, das Sie gerade schreiben. »Wenn Sie den String beibehalten, wird die Zuweisung nicht aufgehoben«, bedeutet also in Wirklichkeit »Wenn eine Instanz der Klasse, an der Sie gerade arbeiten, den String beibehält, wird die Zuweisung nicht aufgehoben.«

Nun folgen die Regeln. (Einzelheiten zur Implementierung stehen in Klammern.)

- Wenn Sie ein Objekt mithilfe einer Methode anlegen, deren Name mit `alloc` oder `new` beginnt oder `copy` enthält, haben Sie den Besitz daran übernommen. (Das gilt unter der Voraussetzung, dass der Beibehaltungszähler des neuen Objekts auf 1 steht und sich das Objekt nicht im Pool zur automatischen Freigabe befindet.) Sie sind dafür zuständig, das Objekt freizugeben, wenn Sie es nicht mehr benötigen. Unter anderem übertragen die Methoden `alloc` (darauf folgt immer eine `init`-Methode), `copy` und `mutableCopy` den Besitz.
- Ein auf *irgendeine* andere Art erstelltes Objekt – zum Beispiel mit einer Komfortmethode – gehört Ihnen *nicht*. (Das gilt unter der Voraussetzung, dass der Beibehaltungszähler des neuen Objekts auf 1 steht und es sich bereits im Pool zur automatischen Freigabe befindet, sodass es zur Zerstörung vorgesehen ist, wenn es nicht vor dem Leeren des Pools beibehalten wird.)
- Wenn Sie nicht Besitzer eines Objekts sind, aber sicherstellen wollen, dass es weiterhin existiert, übernehmen Sie den Besitz, indem Sie die Nachricht `retain` senden. (Dadurch wird der Beibehaltungszähler hoch gesetzt.)
- Wenn Sie Besitzer eines Objekts sind und es nicht mehr benötigen, senden Sie ihm die Nachricht `release` oder `autorelease`. (`release` setzt den Beibehaltungszähler sofort herab. `autorelease` führt dazu, dass die Nachricht `release` gesendet wird, wenn der Pool für die automatische Freigabe geleert wird.)
- Solange ein Objekt mindestens einen Besitzer hat, existiert es weiter. (Wenn sein Beibehaltungszähler auf 0 sinkt, wird ihm die Nachricht `dealloc` gesendet.)

Um die Speicherverwaltung zu meistern, müssen Sie lokal denken. Die Klasse `Possession` braucht nichts über andere Objekte zu wissen, die sich ebenfalls für ihre Instanzvariablen `possessionName` oder `serialNumber` interessieren. Solange eine Instanz von `Possession` Objekte beibehält, die sie behalten möchte, haben Sie kein Problem. Programmierer, die die Sprache noch nicht beherrschen, machen manchmal den Fehler, Objekte im gesamten Verlauf einer Anwendung kontrollieren zu wollen. Tun Sie das nicht. Wenn Sie sich an diese Regeln halten und immer lokal innerhalb einer Klasse denken, brauchen Sie sich keine Sorgen darüber zu machen, was der Rest einer Anwendung mit einem Objekt macht.

Weiter vorn in diesem Kapitel haben wir Änderungen an den Zugriffsmethoden von `Possession` vorgenommen, damit sie die Speicherverwaltung korrekt handhaben. Vorher lief die Anwendung auch hervorragend. Wie kann das sein? Die Objekte, die angelegt und als Instanzvariablen der

`Possession`-Instanzen gesetzt wurden, wurden niemals freigegeben, weshalb es keine Rolle spielte, ob wir sie beibehielten. Echte Anwendungen bestehen jedoch aus viel mehr Einzelteilen, und es werden darin Objekte angelegt und freigegeben. Mit einer ordentlichen Speicherverwaltung hat diese `Possession`-Klasse auch in einer echten Anwendung Bestand.

3.3 Wenn Sie noch mehr wissen wollen: Weiteres zur Speicherverwaltung

Im gesamten Verlauf dieses Buches finden Sie Abschnitte mit dem Titel »Wenn Sie noch mehr wissen wollen« mit eingehenderen Erläuterungen der Themen, die im Kapitel angeschnitten wurden. Diese Abschnitte sind nicht unbedingt wichtig, um Sie dahin zu bringen, wohin Sie wollen, aber wir hoffen, dass Sie sie interessant und hilfreich finden. Da sie mehr in die Tiefe gehen als der restliche Stoff im Kapitel, bewegen sie sich häufig (aber nicht immer) auf höherem Niveau. Fühlen Sie sich beim Lesen eines solchen Abschnitts überfordert, machen Sie sich keine Sorgen, denn das ging noch jedem so, der sich zum ersten Mal mit diesen Themen befasste.

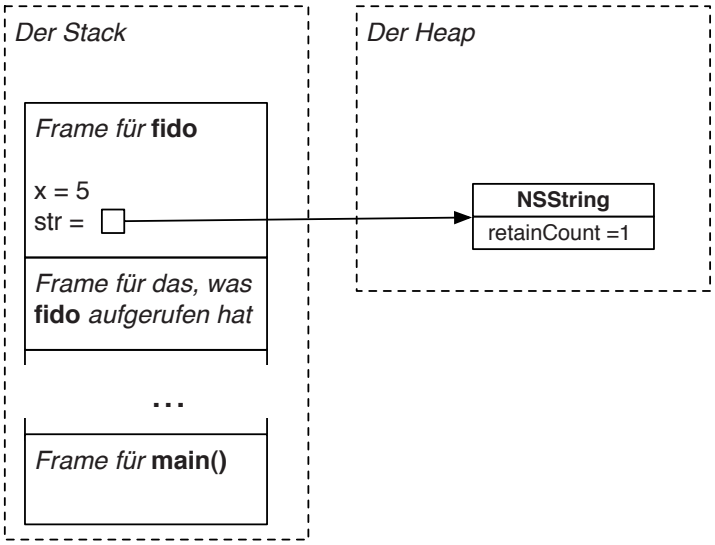
In diesem Kapitel haben wir über den Heap gesprochen und uns angesehen, wie die einzelnen Objekte in diesem Teil des Arbeitsspeichers existieren. Es gibt noch zwei weitere Speicherbereiche für andere Zwecke: das Datensegment und den Stack. Im Datensegment ist die ausführbare Datei der Anwendung untergebracht. Dort befinden sich alle Anweisungen, aus denen Ihre Methoden und Funktionen bestehen. Nach dem Start einer Anwendung ändert sich dieser Speicherbereich nicht mehr. Der Code wird einmal in den Arbeitsspeicher geladen und nicht geändert. Wenn Sie wie folgt ein Literal als `NSString`-Objekt erstellen, befindet sich der Speicher für diesen String ebenfalls im Datensegment:

```
NSString *foo = @"A string";
```

Wir behandeln ihn trotzdem als Objekt, und wenn ein anderes Objekt diesen String haben möchte, muss es ihn kopieren oder beibehalten.

Der Stack ist ein für die Erledigung von Funktionsaufrufen reservierter Speicherbereich. (Eine Methode ist nur eine verkappte Funktion.) Wenn Sie eine Funktion aufrufen, wird ein Teil des Stacks für sie reserviert, der sogenannte *Stackframe*. Der Stackframe hält die Adresse der Funktion fest, die die aktuelle Funktion aufgerufen hat, sodass die aktuelle zur vorherigen Funktion zurückkehren kann, wenn sie abgeschlossen ist. Außerdem enthält der Frame alle der Funktion übergebenen Argumente und Platz für den Rückgabewert der Funktion. Darüber hinaus reserviert er Speicher für die lokalen Variablen der Funktion.

```
- (void)fido
{
   int x = 5;
   NSString *str = [[NSString alloc] init];

   // Die Variable "str" ist eine lokale Variable, die im Stackframe liegt
   // Der String, auf den sie zeigt, befindet sich auf dem Heap.
}
```



▸ *Abbildung 3.6: Zeiger auf dem Stack, Objekt auf dem Heap*

Wenn eine Funktion beendet ist, wird der Stackframe zusammen mit den darin enthaltenen lokalen Variablen zerstört.

Außerdem haben wir zwischen Instanzvariablen, die Zeiger auf Objekte sind, und Primitivelementen unterschieden. Wenn wir einer Klasse Instanzvariablen hinzufügen, erweitern wir die Größe einer Instanz dieser Klasse im Speicher um die Menge Speicher, die die Instanzvariablen für ihre Daten benötigen.

Eine Zeigerinstanzvariable umfasst nur vier Bytes. Das ist die Menge, die für die Adresse eines anderen Objekts auf dem Heap erforderlich ist. Der Inhalt eines Primitivelements wird innerhalb der Instanz abgelegt, sodass zum Beispiel der Datentyp `double` die Größe der Instanz um acht Byte erhöht.

Ein `Possession`-Objekt hat drei Zeigerinstanzvariablen und eine vom Datentyp `int`, die jeweils vier Byte umfassen. Deshalb braucht das Objekt 16 Bytes plus die Anzahl Bytes, die ein `NSObject` belegt. Nehmen wir dafür ebenfalls vier an, dann kommen wir auf 20 Bytes pro `Possession`-Objekt.

Wenn wir Instanzen von `Possession` bilden, werden also 20 Bytes vom Heap zugewiesen. Selbst wenn das `NSString`-Objekt, das der `possessionName` des Objekts ist, 100 Bytes lang ist (weil er 100 Zeichen umfasst), bleibt das Objekt selbst bei einer Größe von 20 Bytes. Das ist für das Verständnis des Unterschieds zwischen Objekten und Zeigern auf Objekte hilfreich.

Copyright