Thema 6: Grundlagen von Propertys

Propertys sind ein Merkmal von Objective-C, das die Kapselung der in einem Objekt enthaltenen Daten ermöglicht. Gewöhnlich enthalten Objekte in Objective-C einen Satz Instanzvariablen, in denen sie die für ihre Arbeit benötigten Daten speichern. Der Zugriff auf diese Instanzvariablen erfolgt normalerweise über Zugriffsmethoden. Mit einer Get-Methode wird die Variable gelesen, mit einer Set-Methode geschrieben. Dieses Konzept wurde standardisiert und in Form von Propertys in Objective-C 2.0 zur Verfügung gestellt. Mithilfe von Propertys kann der Entwickler den Compiler anweisen, automatisch Zugriffsmethoden zu schreiben. Dafür wurde eine neue »Punktschreibweise« eingeführt, um den Zugriff auf die in Klassen gespeicherten Daten kompakter zu machen. Wahrscheinlich haben Sie schon mit Propertys gearbeitet, aber möglicherweise kennen Sie noch nicht alle verfügbaren Optionen. Außerdem sind Ihnen vielleicht nicht alle Feinheiten bekannt. Thema 6 behandelt die Grundlagen von Propertys und stellt deren wichtigste Merkmale vor.

In einer Klasse zur Beschreibung einer Person können der Name der Person, das Geburtsdatum, die Adresse usw. gespeichert sein. Die Instanzvariablen können Sie wie folgt im öffentlichen Interface für die Klasse deklarieren:

```
@interface EOCPerson : NSObject {
@public
   NSString *_firstName;
   NSString *_lastName;
@private
   NSString *_someInternalData;
}
@end
```

Wenn Sie Kenntnisse in Java oder C++ haben, wird Ihnen das vertraut vorkommen. Dort können Sie den Gültigkeitsbereich von Instanzvariablen festlegen. In modernem Objective-C wird diese Technik jedoch nur selten verwendet. Das Problem bei dieser Vorgehensweise besteht darin, dass der Aufbau des Objekts zur Übersetzungszeit definiert wird. Bei jedem Zugriff auf die Variable `_firstName` legt der Compiler den Offset zu dem Speicherbereich fest, in dem das Objekt gespeichert ist. Das funktioniert aber nur so lange, bis Sie eine weitere Instanzvariable hinzufügen. Nehmen wir beispielsweise an, Sie ergänzen vor `_firstName` eine weitere Instanzvariable:

```
@interface EOCPerson : NSObject {
@public
   NSDate *_dateOfBirth;
   NSString *_firstName;
   NSString *_lastName;
```

```
@private
   NSString *_someInternalData;
}
@end
```

Der Offset, der zuvor auf `_firstName` zeigte, verweist nun auf `_dateOfBirth`. Jeder Code, der mit dem festgelegten Offset arbeitet, liest nun den falschen Wert. Zur Veranschaulichung sehen Sie in Abbildung 2–1 das Speicherlayout der Klasse vor und nach der Ergänzung um die Instanzvariable `_dateOfBirth` (wobei die Verwendung von 4-Byte-Zeigern angenommen wird).

Code, der den Offset zur Übersetzungszeit berechnet, versagt bei einer Änderung der Klassendefinition, sofern er dann nicht neu kompiliert wird. Beispielsweise kann es in einer Bibliothek Code geben, der eine alte Klassendefinition verwendet. Wenn er mit Code verlinkt wird, der eine neue Klassendefinition nutzt, kommt es zur Laufzeit zu Inkompatibilitäten. Dieses Problem wird in den verschiedenen Sprachen auf unterschiedliche Weise gelöst. In Objective-C besteht der Ansatz darin, aus Instanzvariablen besondere Variablen zu machen, die von Klassenobjekten festgehalten werden, in denen der Offset gespeichert ist. (Mehr über Klassenobjekte erfahren Sie in Thema 14.) Der Offset wird dann zur Laufzeit nachgeschlagen. Bei einer Änderung der Klassendefinition wird der gespeicherte Offset aktualisiert, sodass bei jedem Zugriff auf die Instanzvariable der korrekte Offset verwendet wird. Es ist sogar möglich, Instanzvariablen zur Laufzeit zu Klassen hinzuzufügen. Man spricht hier von einer stabilen Binärschnittstelle (Application Binary Interface, ABI). Eine ABI definiert unter anderem die Vereinbarungen dafür, wie Code generiert werden soll. Bei einer stabilen ABI können Instanzvariablen auch in einer Klassenerweiterungskategorie (siehe Thema 27) und in der Implementierung definiert werden. Es ist also nicht mehr erforderlich, alle Instanzvariablen im Interface zu deklarieren und damit interne Informationen über die Implementierung in der öffentlichen Schnittstelle preiszugeben.

	Person
+0	_firstName
+4	_lastName
+8	_someInternalData

	Person
+0	_dateOfBirth
+4	_firstName
+8	_lastName
+12	_someInternalData

Abb. 2–1 *Aufbau der Klassendaten vor und nach der Ergänzung um eine weitere Instanzvariable*

Zur Überwindung des Problems trägt auch bei, dass Sie dazu angeregt werden, Zugriffsmethoden zu verwenden, anstatt direkt auf die Instanzvariablen zuzugreifen. Hinter den Propertys stehen zwar nach wie vor die Instanzvariablen, aber sie bilden eine saubere Abstraktion. Zwar könnten Sie selbst Zugriffsmethoden schreiben, doch da die Zugriffsmethoden wie in Objective-C üblich einem strengen Benennungsmuster folgen, war es möglich, ein Sprachkonstrukt einzuführen, um Zugriffsmethoden automatisch erstellen zu lassen. Hier kommt die Syntax mit `@property` ins Spiel.

Propertys verwenden Sie in der Definition eines Objektinterface, um eine Standardmöglichkeit für den Zugriff auf die im Objekt gekapselten Daten bereitzustellen. In diesem Sinne können Sie sich Propertys also als eine Kurzschreibweise dafür vorstellen, dass es Zugriffsmethoden für eine Variable eines gegebenen Typs und Namens geben soll. Betrachten Sie beispielsweise die folgende Klasse:

```
@interface EOCPerson : NSObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

Für einen Benutzer der Klasse ist das gleichbedeutend damit, die Klasse auf die folgende Weise zu schreiben:

```
@interface EOCPerson : NSObject
- (NSString*)firstName;
- (void)setFirstName:(NSString*)firstName;
- (NSString*)lastName;
- (void)setLastName:(NSString*)lastName;
@end
```

Um eine Property zu verwenden, setzen Sie die Punktschreibweise ein – ähnlich wie beim Zugriff auf ein Element in einem dem Stack zugewiesenen `struct` in C. Der Compiler wandelt die Punktschreibweise in Aufrufe der Zugriffsmethoden um, als hätten Sie sie direkt aufgerufen. Daher gibt es keinen Unterscheid zwischen der Punktschreibweise und dem direkten Aufruf der Zugriffsmethoden. Dies können Sie an dem folgenden Codebeispiel erkennen:

```
EOCPerson *aPerson = [Person new];

aPerson.firstName = @"Bob"; // Das Gleiche wie:
[aPerson setFirstName:@"Bob"];

NSString *lastName = aPerson.lastName; // Das Gleiche wie:
NSString *lastName = [aPerson lastName];
```

Das ist aber noch nicht alles, was es zu Propertys zu sagen gibt. Wenn Sie ihn nicht daran hindern, schreibt der Compiler automatisch den Code für die Zugriffsmethoden. Dieser Prozess wird automatische Synthese genannt. Beachten Sie, dass der Compiler dies zur Übersetzungszeit tut, sodass Sie im Editor keinen Quellcode für die synthetisierten Methoden sehen. Der Compiler schreibt aber nicht nur die Zugriffsmethode, sondern fügt der Klasse auch automatisch eine Instanzvariable des erforderlichen Typs hinzu, deren Name aus dem Namen der Property mit vorangestelltem Unterstrich besteht. Im vorhergehenden Beispiel werden also die beiden Instanzvariablen `_firstName` und `_lastName` erstellt. Mit dem Schlüsselwort `@synthesize` in der Implementierungsdatei der Klasse können Sie jedoch Einfluss auf die Benennung dieser Instanzvariablen nehmen:

```
@implementation EOCPerson
@synthesize firstName = _myFirstName;
@synthesize lastName = _myLastName;
@end
```

Wenn Sie diese Syntax verwenden, erhalten Sie Instanzvariablen mit den Namen `_myFirstName` und `_myLastName` statt der Standardbezeichnungen. Es ist allerdings nicht üblich, Instanzvariablen einen anderen als den Standardnamen zu geben. Wenn Sie jedoch kein großer Freund von Unterstrichen bei der Benennung von Instanzvariablen sind, können Sie damit einen beliebigen Namen vergeben. Ich rate Ihnen jedoch dazu, beim Standardschema zu bleiben, da der Code dadurch besser für Personen verständlich ist, die denselben Konventionen folgen.

Wenn Sie nicht wollen, dass der Compiler die Zugriffsmethoden für Sie synthetisiert, können Sie sie auch selbst implementieren. Wenn Sie nur eine der Methoden schreiben, erstellt der Compiler jedoch nach wie vor die andere. Eine andere Möglichkeit, um den Compiler von der Synthese abzuhalten, bietet das Schlüsselwort `@dynamic`, das den Compiler anweist, weder die Instanzvariable zu der Property noch die Zugriffsmethoden automatisch zu erstellen. Beim Kompilieren von Code, der auf die Property zugreift, ignoriert der Compiler dann auch die Tatsache, dass noch keine Zugriffsmethoden definiert sind, und vertraut darauf, dass sie zur Laufzeit zur Verfügung stehen werden. Dieses Verhalten wird beispielsweise ausgenutzt, um Unterklassen von `NSManagedObject` aus CoreData zu erstellen, wobei die Zugriffsmethoden dynamisch zur Laufzeit angelegt werden. Diese Vorgehensweise wird bei `NSManagedObjects` verwendet, da es sich bei den Propertys nicht um Instanzvariablen handelt. Die Daten kommen von dem jeweils verwendeten Datenbank-Back-End. Betrachten Sie dazu das folgende Beispiel:

```
@interface EOCPerson : NSManagedObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

```
@implementation EOCPerson
@dynamic firstName, lastName;
@end
```

In dieser Klasse werden keine Zugriffsmethoden und Instanzvariablen synthetisiert. Der Compiler warnt Sie auch nicht, wenn Sie versuchen, auf eine der Propertys zuzugreifen.

Attribute von Propertys

Ein weiterer Aspekt von Propertys, über den Sie sich im Klaren sein sollten, sind die Attribute, mit denen Sie Einfluss auf die vom Compiler generierten Zugriffsmethoden nehmen können. Im folgenden Beispiel werden drei Attribute verwendet:

```
@property (nonatomic, readwrite, copy) NSString *firstName;
```

Es gibt vier Kategorien von Attributen.

Atomizität

Synthetisierte Zugriffsmethoden enthalten standardmäßig eine Vorkehrung für Sperren, um sie atomar zu machen. Wenn Sie das Attribut `nonatomic` angeben, werden keine Sperren eingesetzt. Beachten Sie, dass es das Attribut `atomic` zwar nicht gibt (der atomare Zustand wird durch das Fehlen des Attributs `nonatomic` angezeigt), es aber trotzdem nicht zu einem Compilerfehler führt, wenn Sie es angeben, um ausdrücklich Atomizität zu verlangen. Wenn Sie die Zugriffsmethoden selbst definieren, müssen Sie für die festgelegte Atomizität sorgen.

Lese- und Schreibzugriff

- **`readwrite`** Es ist sowohl eine Get- als auch eine Set-Methode verfügbar. Wird die Property synthetisiert, generiert der Compiler beide.
- **`readonly`** Es steht nur eine Get-Methode zur Verfügung, und der Compiler generiert bei der Synthese der Property auch nur diese eine Methode. Dieses Attribut können Sie verwenden, wenn Sie eine Property extern nur zum Lesen bereitstellen wollen. In der Klassenerweiterungskategorie können Sie sie dann intern als les- und schreibbar neu deklarieren. Mehr darüber erfahren Sie in Thema 27.

Speicherverwaltung

Propertys kapseln Daten, und für diese Daten wird eine konkrete Semantik für die Besitzverhältnisse gebraucht. Dies betrifft jedoch nur die Set-Methode. Es geht hier beispielsweise um Fragen wie die, ob die Set-Methode den neuen Wert beibehält oder ihn nur der zugrunde liegenden Instanzvariable zuweist. Wenn der

Compiler die Zugriffsmethoden synthetisiert, bestimmt er anhand der folgenden Attribute, was für Code er schreiben soll. Schreiben Sie die Zugriffsmethoden selbst, so müssen Sie sich an die Attribute halten, die Sie in dieser Kategorie festgelegt haben.

- **`assign`** Die Set-Methode ist eine einfache Zuweisungsoperation für skalare Typen wie `CGFloat` und `NSInteger`.
- **`strong`** Die Property definiert ein Besitzverhältnis. Wenn ein neuer Wert gesetzt wird, so wird er zunächst beibehalten. Dann wird der alte Wert freigegeben und der neue gesetzt.
- **`weak`** Die Property definiert eine Beziehung ohne Besitzverhältnis. Wird ein neuer Wert gesetzt, so wird er nicht beibehalten, und der alte Wert wird auch nicht freigegeben. Dies ähnelt der Vorgehensweise bei `assign`, allerdings wird der Wert hier auch in `nil` umgewandelt, wenn das Objekt zerstört wird, auf das die Property zeigt.
- **`unsafe_unretained`** Dies hat dieselbe Bedeutung wie `assign`, wird aber bei Objekttypen verwendet, um eine Beziehung ohne Besitzverhältnis (*unretained*, also ohne Beibehaltung) anzugeben, die unsicher ist (*unsafe*), weil der Wert im Gegensatz zu `weak` bei einer Zerstörung des Ziels nicht in `nil` umgewandelt wird.
- **`copy`** Dies zeigt ein Besitzverhältnis ähnlich wie bei `strong` an, doch anstatt den Wert beizubehalten, wird er kopiert. Das geschieht häufig beim Typ `NSString*`, um die Kapselung zu erhalten, da der an die Set-Methode übergebene Wert eine Instanz der Unterklasse `NSMutableString` sein kann. In diesem Fall könnte der Wert nach dem Setzen der Property verändert sein, ohne dass das Objekt etwas darüber erfährt. Daher wird eine unveränderbare Kopie angelegt, um sicherzustellen, dass sich der String nicht hinter dem Rücken des Objekts ändert. Bei allen veränderbaren Objekten sollte eine Kopie verwendet werden.

Methodennamen

Mit den folgenden Attributen können Sie steuern, welche Namen die Zugriffsmethoden bekommen:

- **`getter=<name>`** Gibt den Namen der Get-Methode an. Dieses Attribut wird häufig für boolesche Propertys verwendet, um dem Namen der Get-Methode ein `is` voranzustellen. Beispielsweise ist in der Klasse `UISwitch` die Property, die angibt, ob der Schalter ein- oder ausgeschaltet ist, wie folgt definiert:

  ```
  @property (nonatomic, getter=isOn) BOOL on;
  ```

- **`setter=<name>`** Gibt den Namen der Set-Methode an. Dieses Attribut wird nur selten genutzt.

Mit diesen Attributen können Sie detailliert Einfluss auf die synthetisierten Zugriffsmethoden nehmen. Beachten Sie aber, dass Sie bei der Implementierung Ihrer eigenen Zugriffsmethoden selbst dafür sorgen müssen, dass sie den angegebenen Attributen gehorchen. Wenn Sie beispielsweise eine Property als `copy` definieren, muss die Set-Methode auch tatsächlich eine Kopie des Objekts verwenden, da die Benutzer der Klasse sonst von falschen Voraussetzungen ausgehen und sich Fehler einschleichen, da die Vereinbarungen nicht eingehalten werden.

Auch in anderen Methoden, die Propertys festlegen, müssen Sie auf die Wahrung der Aspekte achten, die in der Definition der Property angegeben sind. Betrachten Sie als Beispiel eine Erweiterung der Klasse `EOCPerson`, die für die Speicherverwaltung der Property das Attribut `copy` festlegt, da der Wert veränderbar ist. Die Erweiterung umfasst auch einen Initialisierer, der den Anfangswert für den Vor- und Nachnamen festlegt:

```
@interface EOCPerson : NSManagedObject

@property (copy) NSString *firstName;
@property (copy) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
    lastName:(NSString*)lastName;
@end
```

Es ist wichtig, dass Sie sich bei der Implementierung des selbst geschriebenen Initialisierers an das in der Property-Definition angegebene Kopierverfahren halten, weil die Property-Definition die Vereinbarung festlegt, die die Klasse mit den zu setzenden Werten hat. Eine Implementierung des Initialisierers sieht daher wie folgt aus:

```
- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName
{
   if ((self = [super init])) {
      _firstName = [firstName copy];
      _lastName = [lastName copy];
   }
   return self;
}
```

Vielleicht fragen Sie sich, warum Sie stattdessen nicht einfach die Set-Methode der Property nehmen können, die immer sicherstellt, dass die korrekte Art der Speicherverwaltung durchgeführt wird. Wie in Thema 7 weiter ausgeführt wird, sollten Sie in einer `init`- oder `dealloc`-Methode niemals Ihre eigenen Zugriffsmethoden verwenden.

Falls Sie schon Thema 18 gelesen haben, wissen Sie, dass es am besten ist, ein Objekt nach Möglichkeit unveränderbar zu machen. Bei `EOCPerson` müssen Sie dazu beide Propertys schreibgeschützt machen. Der Initialisierer legt die Werte fest, woraufhin sie nicht mehr geändert werden können. In dieser Situation ist es jedoch nach wie vor wichtig, die Speicherverwaltung für die Werte zu deklarieren. Die Property-Definitionen sehen daher wie folgt aus:

```
@property (copy, readonly) NSString *firstName;
@property (copy, readonly) NSString *lastName;
```

Auch wenn keine Set-Methoden erstellt werden, da die Propertys schreibgeschützt sind, müssen Sie die Art der Speicherverwaltung für den Fall angeben, dass der Initialisierer die Werte festlegt. Ohne diese Angabe kann ein Benutzer der Klasse nicht davon ausgehen, dass eine Kopie verwendet wird, weshalb er möglicherweise eine eigene Kopie erstellt, bevor er den Initialisierer aufruft. Das wäre überflüssig und unwirtschaftlich.

Vielleicht fragen Sie sich, worin der Unterschied zwischen `atomic` und `nonatomic` besteht. Wie bereits gesagt, verwenden atomare Zugriffsmethoden Sperren, um die Atomizität sicherzustellen. Wenn dann zwei Threads dieselbe Property lesen und schreiben, ist gewährleistet, dass die Property zu jedem Zeitpunkt einen gültigen Wert hat. Ohne Sperren – also bei der Verwendung von `nonatomic` –, kann der Wert der Property von einem Thread gelesen werden, während ein anderer gerade dabei ist, ihn zu ändern. Wenn das geschieht, kann der gelesene Wert ungültig sein.

Wenn Sie bereits für die iOS-Plattform programmiert haben, ist Ihnen bestimmt aufgefallen, dass alle Propertys als `nonatomic` deklariert werden. Der Grund dafür ist, dass Sperren historisch bedingt einen solchen Zusatzaufwand für iOS verursachen, dass sich ein Leistungsproblem ergibt. Gewöhnlich ist Atomizität auch nicht notwendig, da sie ohnehin keine Threadsicherheit gewährleistet – dazu wäre eine viel tiefgreifendere Art der Sperrung erforderlich. Selbst bei Atomizität kann ein einzelner Thread einen Wert mehrmals unmittelbar hintereinander lesen und dabei unterschiedliche Ergebnisse erhalten, wenn ein anderer Thread gleichzeitig in die Property schreibt. Daher werden in iOS gewöhnlich nicht atomare Propertys verwendet. In OS X stellt der atomare Zugriff auf Propertys normalerweise jedoch keinen Leistungsengpass dar.

Was Sie sich merken sollten

- Mit `@property` können Sie festlegen, welche Daten ein Objekt kapselt.
- Sorgen Sie mit Attributen für den richtigen Umgang mit den gespeicherten Daten.

Was Sie sich merken sollten *(Forts.)*

- Wenn die Instanzvariable, die hinter einer Property steht, an irgendeiner Stelle gesetzt wird, müssen Sie dafür sorgen, dass die für diese Property deklarierten Attribute beachtet werden.
- Verwenden Sie in iOS `nonatomic`, da die Leistung bei `atomic` stark beeinträchtigt wird.

Thema 7: Lesen Sie Instanzvariablen beim internen Zugriff vorzugsweise direkt

Für den externen Zugriff auf die Instanzvariablen eines Objekts sollten Sie immer Propertys verwenden. Wie der interne Zugriff erfolgen sollte, ist in der Objective-C-Gemeinde dagegen ein heiß diskutiertes Thema. Nach Meinung der einen sollten Sie dazu immer Propertys verwenden, nach Meinung der anderen immer einen direkten Zugriff durchführen, und wiederum andere schlagen eine Mischung aus beiden Techniken vor. Ich rate dringend dazu, zum Lesen von Instanzvariablen direkt vorzugehen und zum Setzen Propertys zu verwenden, wobei jedoch einige Warnhinweise zu beachten sind.

Betrachten Sie die folgende Klasse:

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;

// Komfortmethoden für firstName + " " + lastName:
- (NSString*)fullName;
- (void)setFullName:(NSString*)fullName;
@end
```

Die Komfortmethoden `fullName` und `setFullName:` können wie folgt implementiert werden:

```
- (NSString*)fullName {
   return [NSString stringWithFormat:@"%@ %@",
           self.firstName, self.lastName];
}

/** Im Folgenden wird vorausgesetzt, dass der vollständige Name immer aus
* genau zwei Teilen besteht. Die Methode kann aber auch an exotischere Namen
* angepasst werden.
*/
```

```
- (void)setFullName:(NSString*)fullName {
   NSArray *components =
      [fullName componentsSeparatedByString:@" "];
   self.firstName = [components objectAtIndex:0];
   self.lastName = [components objectAtIndex:1];
}
```

Sowohl in der Get- als auch in der Set-Methode greifen wir über die Punktschreibweise für Propertys auf die Instanzvariablen zu. Wenn Sie die beiden Zugriffsmethoden für den direkten Zugriff umschreiben, sehen sie wie folgt aus:

```
- (NSString*)fullName {
   return [NSString stringWithFormat:@"%@ %@",
           _firstName, _lastName];
}

- (void)setFullName:(NSString*)fullName {
   NSArray *components =
      [fullName componentsSeparatedByString:@" "];
   _firstName = [components objectAtIndex:0];
   _lastName = [components objectAtIndex:1];
}
```

Es gibt einige Unterschiede zwischen diesen beiden Varianten:

- Der direkte Zugriff auf die Instanzvariablen ist zweifellos schneller, da er nicht über die Methodenabfertigung von Objective-C läuft (siehe Thema 11). Der Compiler gibt Code aus, der unmittelbar dort auf den Arbeitsspeicher zugreift, wo sich die Instanzvariablen des Objekts befinden.
- Beim direkten Zugriff wird die in der Set-Methode definierte Speicherverwaltung der Property umgangen. Ist die Property beispielsweise als `copy` deklariert, so wird bei einer direkten Festlegung keine Kopie angelegt. Der neue Wert wird beibehalten und der alte freigegeben. (Dies gilt nur bei der Verwendung von ARC, was Sie aber, wie in Thema 30 ausgeführt, nutzen sollten. Ohne ARC wird der alte Wert nicht freigegeben und der neue nicht beibehalten.)
- Beim direkten Zugriff auf Instanzvariablen werden keine Benachrichtigungen zur Schlüssel-Wert-Beobachtung (Key-Value Observing, KVO) ausgelöst. Je nachdem, welches Verhalten Sie von Ihren Objekten verlangen, kann dies möglicherweise zu einem Problem führen.
- Beim Zugriff über Propertys lassen sich Fehler in diesem Zusammenhang leichter finden, da Sie einen Haltepunkt für die Get- oder Set-Methode anlegen können, um zu ermitteln, wer wann auf die Propertys zugreift.

Ein guter Kompromiss besteht darin, Instanzvariablen über die Set-Methode zu schreiben und direkt zu lesen. Dadurch behalten Sie beim Lesen den Geschwindigkeitsvorteil und beim Schreiben die Kontrolle über den Vorgang. Der wichtigste Grund dafür, die Set-Methode zum Schreiben zu verwenden, besteht darin, dass dadurch die vorgesehene Art der Speicherverwaltung erhalten bleibt. Es gibt jedoch einige Fallstricke bei dieser Vorgehensweise.

Das erste Problem tritt auf, wenn Sie Werte in einer Initialisierungsmethode setzen. Hier sollten Sie immer direkt auf die Instanzvariablen zugreifen, da Unterklassen die Set-Methode überschreiben könnten. Nehmen wir an, die Klasse `EOCPerson` hat eine Unterklasse `EOCSmithPerson`, die ausschließlich für Personen mit dem Nachnamen Smith da ist. Diese Unterklasse kann die Set-Methode für `lastName` wie folgt überschreiben:

```
- (void)setLastName:(NSString*)lastName {
    if (![lastName isEqualToString:@"Smith"]) {
        [NSException raise:NSInvalidArgumentException
                    format:@"Last name must be Smith"];
    }
    super.lastName = lastname;
}
```

Es ist denkbar, dass die Basisklasse `EOCPerson` den Nachnamen in ihrem Standardinitialisierer auf einen leeren String setzt. Wenn dies über die Set-Methode erfolgt, wird die Set-Methode der Unterklasse aufgerufen und eine Ausnahme ausgelöst. Es gibt jedoch Situationen, in denen Sie in einem Initialisierer eine Set-Methode verwenden müssen, nämlich wenn die Instanzvariable in der Oberklasse deklariert ist. In diesem Fall ist ein direkter Zugriff auf die Instanzvariable ohnehin nicht möglich, weshalb Sie auf die Set-Methode zurückgreifen müssen.

Auch bei Propertys mit verzögerter Initialisierung ist Vorsicht geboten. In diesem Fall müssen Sie eine Get-Methode einsetzen, da die Instanzvariable sonst niemals initialisiert wird. Nehmen wir an, die Klasse `EOCPerson` hat eine Property, die Zugriff auf ein komplexes Objekt zur Darstellung des Gehirns einer Person gewährt. Wenn nur selten auf diese Property zugegriffen wird und es einen großen Aufwand bedeutet, sie zu setzen, können Sie sie in der Get-Methode wie folgt verzögert initialisieren:

```
- (EOCBrain*)brain {
    if (!_brain) {
        _brain = [Brain new];
    }
    return _brain;
}
```

Wenn Sie nun direkt auf die Instanzvariable zugreifen, bevor die Get-Methode aufgerufen wurde, ist `brain` noch nicht eingerichtet. Daher müssen Sie für jeden Zugriff auf die Property `brain` die Zugriffsmethoden verwenden.

Was Sie sich merken sollten

- Beim internen Zugriff auf Instanzvariablen sollten Sie Daten vorzugsweise direkt lesen und über Propertys schreiben.
- In Initialisierern und in `dealloc` müssen Sie Daten in Instanzvariablen immer direkt lesen und schreiben.
- Wenn Daten verzögert initialisiert werden, müssen Sie sie mithilfe von Propertys lesen.

Thema 8: Wann sind Objekte gleich?

Es ist äußerst nützlich, ermitteln zu können, ob zwei Objekte gleich sind. Eine Prüfung mit dem Operator `==` ergibt jedoch meistens nicht das, was Sie tun wollen, da hierbei die Zeiger miteinander verglichen werden und nicht die Objekte, auf die sie zeigen. Stattdessen sollten Sie die Methode `isEqual:` verwenden, die im Protokoll `NSObject` deklariert ist. Gewöhnlich werden zwei Objekte unterschiedlicher Klasse jedoch immer als ungleich angesehen. Manche Klassen bieten auch besondere Vergleichsmethoden, die Sie einsetzen können, wenn Sie bereits wissen, dass die beiden Objekte, die Sie untersuchen wollen, dieselbe Klasse aufweisen. Betrachten Sie zum Beispiel den folgenden Code:

```
NSString *foo = @"Badger 123";
NSString *bar = [NSString stringWithFormat:@"Badger %i", 123];
BOOL equalA = (foo == bar);              //< equalA = NO
BOOL equalB = [foo isEqual:bar];         //< equalB = YES
BOOL equalC = [foo isEqualToString:bar]; //< equalC = YES
```

Hier können Sie den Unterschied zwischen `==` und den Vergleichsmethoden genau erkennen. `NSString` ist ein Beispiel für eine Klasse, die ihre eigene Vergleichsmethode implementiert, nämlich `isEqualToString:`. Das an diese Methode übergebene Objekt muss ebenfalls ein `NSString` sein, da das Ergebnis ansonsten nicht definiert ist. Diese Methode ist schneller als `isEqual:`, da bei Letzterer die Klasse des zu vergleichenden Objekts nicht bekannt ist und daher einige zusätzliche Schritte erforderlich sind.