



Sebastian
Meyer

Torben
Wichers

2. Auflage

Objective-C 2.0

Programmierung für Mac OS X und iPhone

Teil I

Die Programmiersprache Objective-C 2.0

In diesem Teil:

- **Kapitel 1**
Objekte und Klassen 35
- **Kapitel 2**
Wie werden Nachrichten verarbeitet? 77
- **Kapitel 3**
Kategorien 91
- **Kapitel 4**
Properties 103
- **Kapitel 5**
Protokolle 121
- **Kapitel 6**
Ausnahmebehandlung 143
- **Kapitel 7**
Blöcke 159

Objekte und Klassen

Der größte Unterschied zwischen Objective-C und C ist die Erweiterung um objektorientierte Konzepte. Aber was genau macht die objektorientierte Programmierung aus? An diese Frage wollen wir uns im ersten Abschnitt dieses Kapitels zuerst einmal unabhängig von Objective-C heranwagen. In den Abschnitten 1.2 und 1.3 wollen wir dann zeigen, was Objekte in Objective-C sind und wie sie miteinander interagieren können. In Abschnitt 1.4 auf Seite 44 beschäftigen wir uns dann mit Klassen, um in Abschnitt 1.5 auf Seite 66 mit der Erzeugung von Objekten das Kapitel abzuschließen.

1.1 Was ist objektorientierte Programmierung?

Traditionell unterscheiden prozedurale Programmiersprachen wie C zwischen Daten und den Funktionen, die diese Daten verarbeiten. Die Daten selbst sind so lange statisch und unverändert, bis der Programmcode sie modifiziert. Der Programmcode auf der anderen Seite ist ohne die Daten wenig nützlich. Seine einzige Funktion ist das Verändern der Daten. Funktionen in prozeduralen Programmiersprachen haben keinen eigenen inneren Zustand. Diese Trennung ist begründet in der Art und Weise, wie Computer arbeiten, daher kann sie nicht einfach ignoriert werden. Objektorientierte Programmiersprachen verändern diese Strukturen, indem sie Daten und die zugehörigen Methoden gruppieren. Eine solche Gruppierung ist ein **Objekt** (Engl. **Object**).

Im Zusammenhang mit Objekten spricht man anstelle von Methoden auch gerne von **Nachrichten**, die ein Objekt versteht und die ihm gesendet werden können. Auf diese Nachrichten kann das Objekt dann reagieren und antworten.

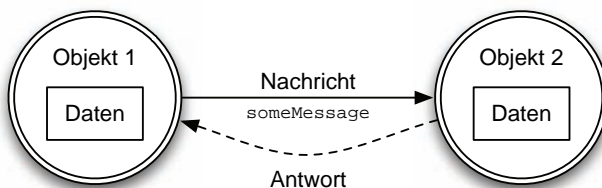


Abb. 1.1: Interaktion von Objekten über Nachrichten

In Abbildung 1.1 sieht man, dass ein Objekt 1 einem Objekt 2 eine Nachricht senden kann. In diesem Fall sendet Objekt 1 eine Nachricht mit dem Namen `someMessage`. Objekt 2 kann dann entscheiden, ob es diese Nachricht versteht und, wenn dies der Fall ist, auf Basis seines internen Zustands (Daten) darauf antworten. Anderenfalls führt das Senden der Nachricht zu einem Fehler. Realisiert wird dieses Bild von Nachrichten natürlich mit Methoden. Eine Nachricht besteht immer aus einer Identifikation (im Beispiel `someMessage`) und evtl. weiteren Argumenten. Für jede von einem Objekt unterstützte Nachricht muss dieses Objekt eine passende Methodenimplementierung zur Abarbeitung bereitstellen.

Die Idee hinter objektorientierten Sprachen ist also, dass ein Objekt sowohl einen Zustand (Daten) besitzt als auch Nachrichten versteht, die diesen Zustand ändern können. Durch die Kombination mehrerer Objekte und die Interaktion dieser untereinander kann ein komplettes Programm erstellt werden. Die beiden Basis-konzepte einer objektorientierten Programmiersprache sind also die Objekte selbst sowie ein Mechanismus, über den die Objekte miteinander über Nachrichten interagieren können. Eine der Ideen hinter objektorientierter Programmierung ist es, über Objekte die reale Welt besser nachbilden zu können.

Als Beispiel soll uns ein Auto dienen: Dieses hat einen inneren Zustand wie zum Beispiel die Füllung des Benzintanks oder die gefahrenen Kilometer sowie eine Möglichkeit, diesen Zustand zu manipulieren, indem es zum Beispiel Benzin verbrennt oder vorwärtsfährt. Wenn man genauer hinschaut, wird man feststellen, dass ein Auto eigentlich nicht ein einzelnes Objekt ist, sondern eine Ansammlung von mehreren kleineren Objekten wie Benzintank oder Motor. Wie fein ein Objekt exakt definiert ist, liegt in der Entscheidung des Betrachters.

Ein Vorteil, den man sich von der objektorientierten Programmierung verspricht, ist die bessere Wiederverwendbarkeit von geschriebenem Programmcode. Durch die Kombination von Objekten kann man das gleiche Objekt in unterschiedlichen Programmen verwenden, wenn ähnliche oder gleiche Probleme gelöst werden müssen. Ein anderer Vorteil ist die bessere Modularisierbarkeit von Programmen. Dadurch, dass man Objekte verwendet, kann man den Programmcode stärker als bisher nach Eigenschaften und Methoden gruppieren und damit eine deutlich stärkere Strukturierung ermöglichen.

David Parnas prägte 1972 den Begriff des **Information Hiding**, der besagt, dass ein Objekt nur das preisgeben soll, was wirklich wichtig zu wissen ist. So soll ein Algorithmus etwa nur angeben, was er tut und welche Daten er benötigt und zurückliefert. Wie er dies tut, ist jedoch für den Benutzer des Algorithmus nicht wichtig, es ist versteckt. Im Folgenden wollen wir uns zwei grundlegende Eigenschaften der Objektorientierung ansehen, die genau dieses Prinzip des Information Hiding umsetzen.

Polymorphismus

Unter **Polymorphismus** (im Deutschen etwa: Vielgestaltigkeit) versteht man die Eigenschaft einer objektorientierten Sprache, dass unterschiedliche Objekte auf die gleiche Nachricht unterschiedlich reagieren können. So kann eine Nachricht **berechneUmfang** eine unterschiedliche Implementierung besitzen, abhängig davon, ob sie zu einem Kreisobjekt oder einem Rechteckobjekt gehört. Aus Sicht des Programmierers können wir die Nachricht bei beiden Objekten aufrufen, ohne deren konkrete Ausgestaltung zu kennen. Ein anderes Beispiel ist die **Addition**. Wenn wir ein Bankobjekt haben, kann dieses sowohl eine Addition für einfache Geldbeträge anbieten als auch eine Addition von Immobilien, für die zuerst ein Wert festgestellt werden muss und möglicherweise noch weitere Faktoren ausschlaggebend sind. Jedoch rufen wir in beiden Fällen die Methode **addiere** auf, die unterschiedlich implementiert ist. Wenn wir den Blick noch einmal auf unser Autobeiispiel lenken, so haben wir auch hier viele Beispiele für Polymorphie: So funktioniert die Möglichkeit zu beschleunigen sowohl bei einem Auto, das auf Benzin basiert, als auch bei einem Auto, das mit einem Elektromotor angetrieben wird, obwohl eventuell dafür intern unterschiedliche Aktionen ausgelöst werden müssen. Auch die Scheinwerferfassungen nehmen Glühlampen unterschiedlicher Hersteller an, solange diese die gleiche Fassung verwenden.

Kapselung von Daten

Ein zweites Konzept zur Umsetzung von Information Hiding ist die Kapselung der Daten. Wie wir wissen, enthält ein Objekt neben Daten auch Methoden, um diese Daten zu verändern. Es liegt nun nahe, den Zugriff auf die internen Daten eines Objekts zu beschränken und stattdessen Methoden anzubieten, die diese Daten setzen oder auslesen. Dies hat mehrere Vorteile. Zum einen ist die Integrität der Daten besser geschützt, wenn man jede Veränderung zuvor auf Gültigkeit prüfen kann und im negativen Fall die Veränderung der Daten unterbinden oder sie auf einen gültigen Zustand setzen kann. Zum anderen müssen Eigenschaften eines Objekts nicht an interne Variablen gebunden sein. Sie können zur Laufzeit beliebig aus anderen Daten berechnet werden. So ist zum Beispiel das Gesamtguthaben, das man bei einer Bank besitzt, die Summe aller Guthaben und Kredite, die dynamisch aus den aktuellen Werten berechnet werden kann.

1.2 Objekte und Dynamic Typing

Wie wir im vorherigen Abschnitt bereits gezeigt haben, nehmen in der objektorientierten Programmierung **Objekte** eine zentrale Rolle ein. Jedes Objekt besitzt einen inneren Zustand (**Instanzvariablen**), der durch eine Menge von Operationen (**Methoden, Nachrichten**) von außen verändert werden kann. Welchen Zugriff ein Objekt über die Methoden von außen auf den internen Zustand gewährt und

wie sich dieser daraufhin verändert, entscheidet das Objekt selbst. Ein Objekt bildet also eine Art Abstraktionsbarriere für die internen Daten. Als Basistyp zur Abbildung von Objekten in Objective-C dient `id`. Hiermit können beliebige Objekte dargestellt werden. Hierzu folgendes kurzes Beispiel:

```
id stringObj = @"Hello, World";
```

In diesem Beispiel wird ein Zeichenketten-Objekt erzeugt und der Variablen `stringObj` zugewiesen. Diese Variable ist vom Datentyp `id`. Der interne Zustand dieses Objekts bildet die Zeichenkette "Hello, World". Zu Zeichenketten, die auf diese Weise erzeugt werden, kommen wir später in Abschnitt 10.3 wieder zurück.

Aber was genau ist `id` nun? Wie wir aus dem vorherigen Abschnitt wissen, ist die Idee bei der objektorientierten Programmierung, über Objekte Elemente bzw. Individuen aus der Realität abzubilden. Aus diesem Grund ist jedes erzeugte Objekt prinzipiell einzigartig. Zur Erstellung eines Objekts nimmt es zur Speicherung seiner internen Daten Hauptspeicher ein und bleibt bis zu seinem Lebensende an dieser Stelle gespeichert. Identifiziert wird ein Objekt also über seine Speicheradresse. Der gesamte Speicherinhalt, den ein Objekt einnimmt, wird also nicht bei jeder Wertzuweisung oder Parameterübergabe wie bei C-Strukturen kopiert. Aus diesem Grund ist der Datentyp `id` eigentlich nichts anderes als ein Pointer, der auf die Speicherstelle des Objekts zeigt. Wir werden später in diesem Kapitel noch einmal darauf zurückkommen, was `id` genau ist.

Als Null-Objekt dient in Objective-C das Schlüsselwort `nil`. Es zeigt äquivalent zu `NULL` aus C für Pointer auf die Speicherstelle `0`, an der kein Objekt abgelegt werden kann. Auch `nil` hat als Datentyp in Objective-C den Datentyp `id`:

```
id someObj = nil;
```

Wie wir später sehen werden, kann der Typ eines Objekts nachträglich auch noch genauer spezifiziert werden. Über diesen Typ kann der Compiler feststellen, welche Methoden ein Objekt versteht, da nicht jedes Objekt gleich ist (siehe hierzu das Konzept der Klassen in Abschnitt 1.4 auf Seite 44). So ist ein Auto kein Fahrrad, oder ein Haus ist nicht gleich einem Boot. Diese zusätzlichen Typinformationen dienen im Grunde jedoch nur zur Unterstützung des Compilers. Ausreichend ist in allen Situationen der Datentyp `id`. Jedoch ist `id` in keiner Weise restriktiv oder schränkt den möglichen Funktionsumfang ein, der mit einem Objekt ausgeführt werden kann. Der Typ bzw. das Verhalten und die dazugehörigen Operationen, die ein Objekt versteht, werden alle zur Laufzeit dynamisch an das Objekt gebunden. Dieses Verhalten nennt sich **Dynamic Typing**. Da eine mögliche Typprüfung auf Basis des Datentyps `id` eher unmöglich ist, kann eine Prüfung im Prinzip erst zur Laufzeit stattfinden. Aus diesem Grund sind alle Prüfungen, die

der Compiler auf Basis der doch vorhandenen Typen durchführt, eher als Hinweise und Warnungen zu verstehen und müssen nicht immer korrekt sein. Damit der Typ eines Objekts zur Laufzeit geprüft werden kann, besitzt jedes Objekt die Instanzvariable `isa`. Diese Instanzvariable zeigt auf ein anderes Objekt, das den Typ (**Klasse**) und damit die Metainfo zu einem Objekt repräsentiert. Wie genau es zu dieser Klasseneinteilung von Objekten kommt, sehen wir später in Abschnitt 1.4 auf Seite 44.

Das Dynamic Typing und die Auflösung des Typs über die Instanzvariable `isa` sowie die Metainformationen, die damit zur Laufzeit über ein Objekt zur Verfügung stehen, bieten dem Programmierer die Möglichkeit, zur Laufzeit auf diese Informationen zuzugreifen und abhängig davon zu reagieren. Diese Möglichkeit, sich zur Laufzeit über die Metainformationen zu informieren, wird auch als **Introspektion** bezeichnet. Der Umfang der Selbstauskunft, die Objective-C einem Programmierer bietet, geht sehr weit und wird später in Kapitel 13 im Detail erklärt.

1.3 Nachrichten und Dynamic Binding

Neben dem Konzept der Objekte gehört die Idee, dass diese Objekte untereinander Nachrichten austauschen, zu den Grundkonzepten der objektorientierten Programmierung. Wir senden einem Objekt eine Nachricht, auf die dieses Objekt uns basierend auf seinem internen Zustand antwortet und diesen eventuell ändert. Jedes Objekt versteht eine andere Menge von Nachrichten. Die Nachrichten stellen die Abstraktionsbarriere zwischen der Außenwelt und dem internen Zustand eines Objekts dar.

Im Prinzip ist das Nachrichtenkonzept nichts anderes als der Aufruf einer Funktion und die Antwort ist der Rückgabewert dieser Funktion. Dabei kann die Funktion auf die Instanzvariablen des Objekts zugreifen und diese eventuell auch ändern. Jedoch kann diese exakte Methode, die hinter einem Aufruf steckt, erst zur Laufzeit festgestellt werden. Dazu allerdings später.

Senden von Nachrichten

Wir wollen uns nun erst einmal ansehen, wie wir eine solche Nachricht an ein Objekt schicken. Zur Unterscheidung von einfachen C-Funktionsaufrufen wurde in Objective-C eine komplett neue Syntax eingeführt. Diese Syntax für Methodenaufrufe lehnt sich an die Syntax der Programmiersprache SmallTalk an.

Um einen Nachrichtenaufruf stehen immer eckige Klammern. Als erstes Element in den eckigen Klammern steht der Ausdruck, der das Objekt berechnet, an den die Nachricht geschickt werden soll. Darauf folgen der Name der Funktion und die Argumente.

Die einfachsten Formen von Nachrichten bzw. Methoden besitzen keine zusätzlichen Argumente. Auf den Ausdruck für das Objekt folgt also nur der Name der Methode:

```
id strObj = @"Hello, World";  
int length = [strObj length];
```

In diesem Fall wird an das Zeichenketten-Objekt `strObj` die Nachricht `length` gesendet. Dieses Objekt gibt durch diese Methode die Länge seiner internen Zeichenkette zurück. Die Implementierung der Methode `length` hat also Zugriff auf das Objekt `strObj` und kann damit auf alle seine Instanzenvariablen zugreifen. Eine Nachricht kann äquivalent zu C-Funktionen auch keinen Wert als Ergebnis liefern. In diesem Fall ist der Rückgabetypp der Nachricht ebenfalls `void`.

Es gibt aber auch Nachrichten mit zusätzlichen Argumenten. Hierzu zuerst der Aufruf einer Methode mit einem Argument:

```
unichar ch = [strObj characterAtIndex: 4];
```

Bei einem Argument folgt auf das Objekt und die Methode noch das Argument, das dem Methodenaufruf mitgegeben wird. In diesem Beispiel wird von der Zeichenkette hinter dem Objekt `strObj` das vierte Zeichen beginnend bei 0 abgefragt, also das `o`. Als Hinweis auf ein Argument endet der Methodename mit einem Doppelpunkt. Der Doppelpunkt hinter dem `characterAtIndex` gehört zum Namen der Methode. Es handelt sich allerdings nicht nur um eine Konvention, sondern gehört zur Syntax und kann auch somit nicht entfallen.

Bei mehr als einem Argument folgen nun nicht einfach mehrere Doppelpunkte gefolgt von der entsprechenden Anzahl der Argumente, sondern der Name der Nachricht wird in einzelne Teile zerlegt, um jeweils mit einem Doppelpunkt endend die einzelnen Argumente zu benennen. Für zwei Argumente sieht das Ganze dann wie folgt aus:

```
id str = @"Hello, World";  
if ([str compare:@"hello, world" options:NSCaseInsensitiveSearch] == 0) {  
    NSLog(@"Die Zeichenketten sind bis auf Groß-/Kleinschreibung gleich");  
}
```

In diesem Beispiel wird das Objekt `str` mit dem Objekt für die Zeichenkette `@"hello, world"` verglichen. Als erstes Argument wird die Vergleichszeichenkette übergeben. Als zweites Argument mit dem Namen `options`: wird die Konstante `NSCaseInsensitiveSearch` übergeben, wodurch beim Vergleich der Zeichenketten die Groß- und Kleinschreibung nicht beachtet wird. Der Name der

Methode setzt sich aus allen Einzelstücken inklusive der Doppelpunkte zusammen. In diesem Fall lautet der Name also `compare:options:`.

Für noch mehr Argumente setzt sich dieses Muster entsprechend fort. Die einzelnen Abschnitte des Namens vor den Argumenten dienen als eine Art benannte Argumente, wodurch genauer erkennbar wird, welche Aufgaben die einzelnen Argumente besitzen. Im Gegensatz zu Programmiersprachen wie C++ oder Java gibt es dafür keine überladenen Funktionen.

Den Vorteil zwischen den benannten Parametern des Objective-C-Methodenaufrufs und dem Funktionsaufruf von C ohne benannte Parameter wollen wir im folgenden Beispiel noch etwas verdeutlichen:

```
StringCompareWithOptions(str, "hello, world", NSCaseInsensitiveSearch);  
[str compare: "hello, world" options: NSCaseInsensitiveSearch];
```

Hier sieht man sehr schön, dass die Beschreibung der Funktion inkl. Argumentnamen bei C vorne im Namen der Funktion stehen und hinten als kommaseparierte Liste die Argumente folgen. Ein echter Bezug zwischen Argumenten und Namen existiert also nicht. Bei dem Objective-C-Methodenaufruf sieht es hingegen anders aus. Hier sieht man sehr deutlich, welche Aufgabe die einzelnen Argumente übernehmen. Noch deutlicher wird das Ganze, wenn man die in C-Programmen üblichen sehr kryptischen und kurzen Funktionsnamen verwendet. In diesem Fall ist eine Deutung der Argumente nur noch über den Funktionsablauf oder eine zusätzliche Dokumentation möglich.

Hinweis

Da der Doppelpunkt zum Namen gehört, wäre als Name einer Funktion oder eines Arguments auch nur der Doppelpunkt denkbar. Hiermit würde jedoch der Vorteil der Benennung der Argumente verloren gehen, wodurch die Aufgabe einzelner Argumente am Namen nicht mehr zu erkennen ist.

Es gibt allerdings auch Methoden, die eine variable Anzahl an Argumenten annehmen. Beim Aufruf solcher Methoden übergibt man dem letzten Argumentnamen eine Liste von Ausdrücken als Argumente. Eine formatierte Zeichenkette lässt sich dann zum Beispiel über die Funktion `stringWithFormat:` wie folgt erzeugen:

```
id strObj = [NSString stringWithFormat:@"a = %f, b = %i", 3.14, 25 + 7];
```

Die drei Argumente `"a = %f, b = %i", 3.14` und `25 + 7` werden als kommaseparierte Liste dem letzten und einzigen Argument übergeben. Die Kommata selbst gehören im Gegensatz zum Doppelpunkt dann allerdings nicht zum Namen der Methode.

Die Aufrufe von Methoden dürfen natürlich auch beliebig geschachtelt werden. So dürfen sowohl das Argument, an das eine Nachricht geschickt wird, als auch die Argumente einer Nachricht selbst sich wieder aus einem Aufruf einer Nachricht ergeben:

```
int len = [[NSString stringWithFormat:@"Pi = %f", 3.14] length];  
[obj1 someMessage: [obj2 someOtherMessage] argument: 22 - 7];
```

Wichtig

Durch die Zerteilung des Methodennamens in einzelne Teile zur Benennung der Argumente könnte man auf die Idee kommen, dass man einige Argumente auch weglassen könnte oder die Reihenfolge der Argumente beliebig wäre. Dies ist jedoch nicht erlaubt. Ein Methodenaufruf muss immer inklusive aller Argumente in der richtigen Reihenfolge geschehen. In manchen Situationen scheint es allerdings so, als ob Argumente doch wegfallen dürften, da in Programmen Methoden mit ähnlichen Namen, aber unterschiedlichen Argumenten verwendet werden. Hierbei handelt es sich dann allerdings auch wirklich um unterschiedliche Methoden. So gibt es zu der oben verwendeten Methode `compare:options:` bei Cocoa-Zeichenketten auch die Methode `compare:` ohne das Argument `options:`.

Es gibt seit Objective-C 2.0 noch eine weitere Form des Methodenaufrufs über den Punkt-Operator (`.`). Mit dieser Art des Methodenaufrufs werden wir uns in Abschnitt 4.3 genauer beschäftigen.

Dynamic Binding und Polymorphismus

Wie wir gezeigt haben, unterscheiden sich Nachrichten oder Methodenaufrufe auf Objekten stark von den klassischen Funktionsaufrufen der Programmiersprache C. Neben den Unterschieden in der Syntax ist der größte Unterschied, dass sich eine Nachricht immer auf ein Objekt bezieht. Dieses Objekt entscheidet auf Basis des Methodennamens dann, welche implementierte Funktion in Wirklichkeit zur Bearbeitung dieser Nachricht ausgeführt wird. Die Tatsache, dass sich unterschiedliche Objekte auf ein und dieselbe Nachricht völlig verschieden verhalten können, nennt man auch **Polymorphismus**. So reagiert zum Beispiel ein Objekt, das einen Kreis darstellt, auf eine Methode `display` zum Zeichnen des Objekts ganz anders als ein Objekt, das ein Rechteck darstellt, obwohl beide die Methode `display` verstehen. Durch den Polymorphismus der Methodenaufrufe ändert sich die Art, wie man eine objektorientierte Sprache programmiert, erheblich. In Programmiersprachen wie SmallTalk kann aufgrund des Polymorphismus sogar ganz auf Steuerstrukturen wie `if`-Anweisungen oder Schleifen verzichtet werden.

Ermöglicht wird der Polymorphismus in Objective-C erst durch die Unterscheidung zwischen Methodennamen und Implementierung. Aus diesem Grund gibt es in Objective-C für Methodennamen, auch **Selektor** genannt, den Datentyp SEL und für die Implementierung den Datentyp IMP. Für jeden Selektor kann es in einem Programm also eine Vielzahl an verschiedenen Implementierungen geben. Für die Methode `compare:options:` bekommt man den Selektor zum Beispiel durch folgende Anweisung:

```
SEL methodSelector = @selector(compare:options:);
```

Über einen solchen Selektor wird dann bei einem Methodenaufruf die passende Implementierung für das entsprechende Objekt zur Laufzeit gesucht. Man bezeichnet diesen Suchvorgang äquivalent zum Dynamic Typing auch als dynamische Bindung (**Dynamic Binding**). Wie genau diese Suche in Objective-C eingebaut ist und wie sie funktioniert, wollen wir uns später in Kapitel 2 ansehen. Bis jetzt reicht uns erst einmal, dass es eine Unterscheidung zwischen Selektor und Implementierung gibt und dass Methodenaufrufe dynamisch zur Laufzeit an die korrekte Implementierung gebunden werden, um Polymorphismus zu ermöglichen.

Sonderrolle des Null-Objekts »nil«

Eine Sonderrolle im Bezug auf Methodenaufrufe nimmt das Null-Objekt `nil` ein. Im Gegensatz zu vielen anderen Programmiersprachen versteht das Null-Objekt jede beliebige Nachricht, die man ihm schickt. Für den Fall, dass die Methode eine Zahl als Ergebnis erwartet, liefert das Null-Objekt die Zahl 0 in dem entsprechenden Datentyp zurück. Wird ein Objekt oder ein Pointer als Ergebnis erwartet, zeigt das Ergebnis auf die Speicheradresse 0.

Hierzu das folgende Beispiel mit der Ausgabe in Abbildung 1.2:

```
id objVar = nil;
int intVar = [objVar someMethod: 5 + 7];
id idVar = [objVar someOtherMethod: nil arg: 6];

NSLog(@"intVar = %i", intVar);
NSLog(@"objectVar = %@", idVar);
```

Listing 1.1: Senden von Nachrichten an `nil`

```
Meldung
intVar = 0
objectVar = (null)
```

Abb. 1.2: Senden von Nachrichten an das Null-Objekt

Diese Sonderrolle erlaubt es, an vielen Stellen im Programm sehr viel einfacher Programme zu schreiben. Andernfalls müsste man vor jeder Verwendung eines Objekts prüfen, ob es das Null-Objekt ist:

```
if (objVar != nil) {  
    [objVar someMethod];  
}
```

In Programmen vieler anderer Programmiersprachen sieht man sehr häufig solche Aufrufe. Dies hat damit zu tun, dass Null-Objekte in anderen Programmiersprachen einen Ausnahmestand darstellen. Der Versuch, einem Null-Objekt eine Nachricht zu senden, resultiert dann in einem Fehler. Da `nil` in Objective-C keine Ausnahmesituation darstellt, sondern ein normales Objekt ist, kann auf diese Abfrage verzichtet werden. Durch den Wegfall dieser Abfragen erhöht sich die Lesbarkeit von Programmen an vielen Stellen enorm. Für den Fall, dass ein Objekt allerdings für sehr viele Methodenaufrufe hintereinander benötigt wird, kann sich jedoch aus Performancegründen eine vorherige Abfrage auf `nil` trotzdem lohnen.

1.4 Klassen

In den vorherigen Abschnitten haben wir gezeigt, dass bei Objective-C die Objekte eine zentrale Rolle einnehmen. Diesen Objekten können wir Nachrichten schicken, auf die sie reagieren. Das Verhalten eines Objekts steckt also in der Implementierung der unterstützten Nachrichten. Da es allerdings viel zu aufwändig wäre, wirklich für jedes Objekt das Verhalten genau zu definieren, bedient sich Objective-C dem Konzept der **Klassen**. Eine Klasse stellt eine Sammlung von Objekten dar, die das gleiche Verhalten besitzen, also eine Art Prototyp für ein späteres Objekt. Als Programmierer muss man nur das Verhalten dieser ganzen Klasse beschreiben, um das Verhalten aller Objekte (**Instanzen**) dieser Klasse zu bestimmen. Zur Identifizierung besitzt jede Klasse einen Namen. So beschreibt zum Beispiel die Klasse `NSString` im Cocoa-Framework alle Objekte, die eine Zeichenkette darstellen.

Die Namen beginnen per Konvention immer mit einem großen Buchstaben. Falls der Name aus mehreren Teilwörtern besteht, wie zum Beispiel `BankAccount`, wird jeweils der erste Buchstabe der Teilwörter großgeschrieben. Alle anderen Buchstaben sind ebenfalls klein. Sollten Teilwörter der Klasse zu einzelnen Buchstaben abgekürzt werden, wie bei `NSString` das `NS` (**NextStep**), so werden diese ebenfalls großgeschrieben. Diese Art der Namenskonvention nennt man auch **Camel Case** oder **Binnengroßschreibung**, aufgrund der immer wieder auftauchenden Großbuchstaben in einem sonst kleingeschriebenen Wort.

Allerdings sind in Objective-C auch Klassen wiederum Objekte. Für jede Klasse generiert der Compiler genau ein **Klassen-Objekt** je Klasse, auf das wir durch den Namen der Klasse zugreifen können. An dieses Objekt können wir natürlich auch wieder Nachrichten schicken. So kann zum Beispiel durch Senden der Methode `new` an eine Klasse eine Instanz dieser Klasse erzeugt werden:

```
id strObj = [NSString new];  
id myObj = [MyClass new];
```

Klassen-Objekte

Für die Klassen-Objekte gibt es den eigenen Datentyp `Class`. Um von einem Objekt die Klasse zu erfahren, gibt es die Methode `class`:

```
id obj = @"abcd";  
Class objClass = [obj class];
```

In diesem Klassen-Objekt sind Metainformationen über die Klasse gespeichert. Zu diesen Metainformationen gehört zum einen der Name der Klasse (`[objClass className]`) sowie die Superklasse als Klassen-Objekt (`[objClass superclass]`). Es beinhaltet aber auch eine Auflistung aller Instanzvariablen der Klasse sowie der Methoden inklusive der Typsignaturen, die durch die Klasse implementiert sind. Mit diesen Klassen-Objekten und der Auflistung der Methoden wird die dynamische Bindung zur Laufzeit durchgeführt (siehe Kapitel 2). Diese Metainformationen ermöglichen auch viele der Introspektionsmöglichkeiten von Objective-C. Interessant ist, dass viele der Informationen und damit das Verhalten eines Objekts auch zur Laufzeit verändert werden kann (siehe Kapitel 13).

Hinweis

Für jedes Klassen-Objekt, das der Compiler generiert, wird zusätzlich ein Metaklassen-Objekt generiert, das wiederum die Metainformationen für das Klassen-Objekt beinhaltet. Für dieses Metaklassen-Objekt gibt es kein weiteres Metaklassen-Objekt. Das Metaklassen-Objekt wird allerdings nur intern vom Laufzeitsystem verwendet.

Zugriff auf das Klassen-Objekt erhält man auch direkt über den Namen der Klasse, indem man diesem die Nachricht `class` sendet:

```
Class strClass = [NSString class];
```

Wenn man in diesem Klassen-Objekt eine Nachricht schicken möchte, kann man dieses auch direkt erledigen, ohne sich zuerst das Klassen-Objekt mit der Nachricht `class` zu holen, das heißt, der Ausdruck `[NSString className]` ist äquiva-

lent zu `[[NSString className] className]`. Will man das Klassen-Objekt aber zum Beispiel in einer Variablen speichern (wie oben) oder als Argument einer Methode übergeben, so kann nicht nur der Klassenname verwendet werden. Der folgende Ausdruck erzeugt also einen Übersetzungsfehler:

```
Class strClass = NSString;
```

1.4.1 Static Typing als Alternative zum Dynamic Typing

Zur dynamischen Typisierung über den Datentyp `id` gibt es als Alternative die Möglichkeit, den Typ eines Objekts auch genauer zu beschreiben. Hierzu dient die Zugehörigkeit zu einer Klasse als Typzuordnung. Die Klasse selbst nimmt in diesem Fall die Rolle des Datentyps ein. Die Identifizierung des Typs geschieht ebenfalls über den Namen der Klasse. Allerdings handelt es sich dadurch nicht automatisch um einen Pointer wie bei der Verwendung von `id`. Da eine Zeichenkette in Cocoa über die Klasse `NSString` dargestellt wird, ist ein Objekt der Klasse auch vom Typ `NSString*`:

```
NSString* strObj = @"Hallo, World";
```

Durch diese Art der Typisierung von Variablen bekommt der Compiler zusätzliche Informationen über das Verhalten bzw. die Instanzvariablen und unterstützten Methoden des möglichen Objekts, auf das die Variable zeigt. Mit diesen zusätzlichen Informationen kann der Compiler bereits zum Zeitpunkt des Übersetzens die Typen bei der Verwendung dieser Variablen prüfen. Somit können zum Beispiel Methodenaufrufe geprüft und, falls eine Methode von einem Objekt des entsprechenden Typs bzw. der entsprechenden Klasse nicht unterstützt wird, eine Warnung ausgegeben werden. Diese Form der zusätzlichen Typisierung und Nutzung der Typinformationen zur Übersetzungszeit nennt man auch statische Typisierung (**Static Typing**).

Um möglichst viele Fehler eines Programms zu entdecken, die sich über Typen von Objekten erschließen lassen, ist es sinnvoll, soweit möglich, die statische Typisierung über `id` vorzuziehen. Außerdem erhöht die statische Typisierung oft die Lesbarkeit eines Programms erheblich, da sich die Bedeutung einer Variablen durch die Angabe ihrer Klasse viel besser erschließt.

Es gibt allerdings auch Situationen, in denen eine statische Typisierung nicht möglich ist. Wenn zum Beispiel nur ein beliebiges Objekt erwartet wird und sein Typ egal ist, so muss hierfür `id` verwendet werden. Ein Beispiel für solche Klassen, in denen die Verwendung von `id` nicht zu umgehen ist, sind die Klassen der Collection-API (siehe Kapitel 11). Sie dienen als Container-Klasse, um andere Objekte aufzunehmen, zum Beispiel zur Darstellung von Mengen. Diese Mengen dürfen dann beliebige Objekte, also Objekte vom Typ `id`, beinhalten. Ein anderes Beispiel

ist die Methode `new`, die wir im vorherigen Abschnitt bereits vorgestellt haben. Diese Methode liefert eine neue Instanz einer Klasse zurück. Der Rückgabewert dieser Methode ist also von der entsprechenden Klasse abhängig. Da diese Methode allerdings für alle Klassen nur einmal implementiert ist, ist ihr Ergebnistyp `id`. Man kann den Wert allerdings sofort einer Variablen des entsprechenden Typs zuweisen:

```
MyClass* myObj = [MyClass new];
```

Genau diese Mischung aus dynamischer und statischer Typisierung stellt für den Compiler ein Problem dar, da an all den Stellen, an denen `id` verwendet wird, keine Informationen über den Typ vorhanden sind. Die Folge hieraus ist, dass die Typprüfungen, die der Compiler zur Übersetzungszeit durchführt, eventuell falsch sind. Aus diesem Grund werden die erkannten Typfehler auch nur als Warnungen und nicht als Fehler ausgegeben. Es kann also sein, dass Sie ein korrektes Programm schreiben, der Compiler allerdings trotzdem eine Warnung meldet. So führt zum Beispiel jeder Nachrichtenaufruf an eine Variable vom Typ `id` zu einer Warnung, da nicht sichergestellt werden kann, dass das Objekt diese Methode auch wirklich versteht.

1.4.2 Aufbau einer Klasse

Nachdem Sie nun bereits eine ganze Menge über Objekte, Methoden, Typen und Klassen gelesen haben, wollen wir uns nun einmal anschauen, wie wir unsere eigenen Klassen implementieren. Hierzu werden ähnlich zu Java oder C++ die Definitionen einer Klasse, das **Klasseninterface**, von den eigentlichen Implementierungen getrennt. Das Klasseninterface, mit Angabe der Instanzvariablen und der unterstützten Methoden, wird normalerweise in der `.h`-Datei formuliert. Die eigentliche Implementierung der Methoden geschieht dann in der `.m`-Datei. Die Aufteilung in zwei Dateien ist wie in C für den Compiler jedoch nicht notwendig. Allerdings dienen die `.h`-Dateien dazu, mehreren getrennten Programmteilen die darin definierten Klassen bekannt zu machen. Übersetzt werden später nur die `.m`-Dateien, die sich auf die `.h`-Dateien beziehen. Dazu allerdings später. Versuchen wir nun, eine Klasse zu entwickeln, über die wir ein Bankkonto abbilden können. Gucken wir uns zuerst an, wie ein Klasseninterface aufgebaut ist:

```
@interface BankAccount : NSObject {
    // Instanzvariablen
}
// Methodendeklarationen
@end
```

Listing 1.2: Struktureller Aufbau eines Klasseninterface

Ein Klasseninterface beginnt immer mit `@interface` und endet mit `@end`. Typischerweise steht dieses Klasseninterface in einer `.h`-Datei, die nach dem Namen der Klasse benannt ist, also `BankAccount.h`.

Der Name der neuen Klasse lautet `BankAccount`. Hinter dem Namen der Klasse folgt durch einen Doppelpunkt getrennt ihre Superklasse. In diesem Fall ist die Superklasse `NSObject`. Was dies genau bedeutet, wird im Abschnitt 1.4.3 auf Seite 51 beschrieben. Die Definition einer Superklasse kann auch entfallen, was allerdings nicht zu empfehlen ist. In Abschnitt 9.3.6 zeigen wir ein Beispiel für eine Klasse ohne Oberklasse. Bis jetzt wollen wir die Angabe der Superklasse erst einmal ignorieren.

Auf den Namen folgt in geschweiften Klammern die Definition der Instanzvariablen. Die Werte dieser Instanzvariablen bilden hinterher den internen Zustand einer Instanz dieser Klasse. Ein Beispiel für Instanzvariablen könnte wie folgt aussehen:

```
NSString* name;  
int accountNumber;  
float balance;
```

Listing 1.3: Instanzvariablen der Klasse `BankAccount`

In diesem Fall gibt es drei Instanzvariablen: eine für den Namen des Kontoinhabers (`name`) als Zeichenkette, eine für die Kontonummer (`accountNumber`) und eine für den Kontostand (`balance`). Auf das Thema Instanzvariablen gehen wir in Abschnitt 1.4.4 auf Seite 54 in aller Ausführlichkeit ein.

Zuletzt folgen die Methoden oder Nachrichtendeklarationen. Hier können sowohl die Nachrichten spezifiziert werden, die eine Instanz versteht, als auch die Methoden, die wir dem entsprechenden Klassen-Objekt schicken dürfen (**Klassenmethoden**). Beispiele hierfür sind folgende:

```
- (id) initWithNumber: (int) number andName: (NSString*) name;  
+ (BankAccount*) accountWithNumber: (int) number andName: (NSString*) name;  
  
- (int) accountNumber;  
- (NSString*) name;  
- (float) deposit: (float) money;  
- (float) withdraw: (float) money;  
- (float) balance;
```

Listing 1.4: Methoden der Klasse `BankAccount`

Bei den Methoden `accountNumber`, `name`, `deposit:`, `withdraw:`, `balance` und `initWithNumber:andName:`, die durch das Minus (-) gekennzeichnet sind, handelt es sich um Methoden, die ein Objekt dieser Klasse versteht. Das Plus (+) zu

Beginn von `accountWithNumber:andName:` kennzeichnet diese Methode als Klassenmethode, also als Methode, die das Klassen-Objekt versteht. Insgesamt sieht die Definition der Klasse `BankAccount` wie folgt aus:

```
@interface BankAccount : NSObject {
    NSString* name;
    int accountNumber;
    float balance;
}
- (id) initWithNumber: (int) number andName: (NSString*) name;
+ (id) accountWithNumber: (int) number andName: (NSString*) name;

- (int) accountNumber;
- (NSString*) name;
- (float) deposit: (float) money;
- (float) withdraw: (float) money;
- (float) balance;
@end
```

Listing 1.5: Vollständiges Klasseninterface der Klasse `BankAccount`

Die eigentliche Implementierung der Methode erfolgt dann in der entsprechenden `.m`-Datei, also in diesem Fall zum Beispiel `BankAccount.m`. Die Implementierung einer Klasse ist sehr ähnlich zu einem Klasseninterface aufgebaut:

```
@implementation BankAccount : NSObject {
    // Instanzvariablen
}
// Methodenimplementierungen
@end
```

Listing 1.6: Struktureller Aufbau einer Klassen-Implementierung

Es unterscheidet sich nur darin, dass es nicht mit `@interface`, sondern mit `@implementation` beginnt. Außerdem sind nicht nur die Methodensignaturen angegeben, sondern die gesamten Implementierungen der Methoden. Die Implementierung für die Methode `deposit:` zum Einzahlen von Geld sieht zum Beispiel wie folgt aus:

```
- (float) deposit: (float) money {
    if (money < 0) return balance;
    balance += money;
    return balance;
}
```

Listing 1.7: Implementierung der Methode `deposit:`

In dieser wird das Einzahlen von Geld nur dann erlaubt, wenn ein positiver Betrag `money` eingezahlt werden soll. Ist dies der Fall, wird der Betrag zum Kontostand in der Instanzvariablen `balance` aufaddiert. Als Ergebnis liefert die Funktion den neuen Kontostand.

Da jedoch eine Implementierung sich immer auf ein Klasseninterface beziehen muss und nicht von dessen Instanzvariablen, Namen und Superklasse abweichen darf, sind viele der Informationen redundant. Um sich auf das Klasseninterface zu beziehen, müssen wir in der `BankAccount.m`-Datei die `BankAccount.h`-Datei importieren. Hierzu steht vor der Implementierung diese Zeile:

```
#import "BankAccount.h"
```

Aus diesem Grund können sowohl Superklasse als auch Instanzvariablen entfallen und die Implementierung sieht wie folgt aus:

```
@implementation BankAccount  
// Methodenimplementierungen  
@end
```

Listing 1.8: Verkürzte Struktur einer Klassen-Implementierung

Eine Instanz der Klasse `BankAccount` kann dann wie folgt erzeugt werden:

```
BankAccount *account = [BankAccount accountWithNumber: 12345678  
                        andName: @"Max Mustermann"];
```

In Abschnitt 1.4.5 auf Seite 60 beschäftigen wir uns noch einmal genauer mit der Definition und der Implementierung von Methoden.

Verwendung von anderen Klassen

Bei der Definition von Klassen kommt es sehr häufig vor, dass andere Klassen für Instanzvariablen oder als Rückgabewert oder Argumenttyp von Methoden verwendet werden. Um in diesen Situationen als Datentyp nicht einfach nur `id` zu schreiben, muss die Definition der Klasse bekannt sein. Nehmen wir als Beispiel an, dass wir in unserer Klasse `BankAccount` für eine Instanzvariable `bank` die Klasse `Bank` verwenden:

```
@interface BankAccount : NSObject {  
    Bank* bank;  
    // ...  
}  
// ...  
@end
```

Listing 1.9: Verwendung der Klasse `Bank`

Gehen wir außerdem davon aus, dass die Klasse `Bank` in der Datei `Bank.h` definiert wurde. Eine Möglichkeit ist nun einfach, vor dem Klasseninterface die Datei `Bank.h` zu importieren:

```
#import "Bank.h"
```

Allerdings kann es passieren, dass diese Klasse ebenfalls die Klasse `BankAccount` verwendet oder eine Klasse, die `BankAccount` verwendet, oder so weiter. In diesem Fall würde sich ein zyklischer Import bilden.

Uns würde es aber reichen, wenn wir dem Compiler in der `.h`-Datei nur das Versprechen geben, dass es die entsprechende Klasse `Bank` gibt, und erst in der `.m`-Datei, in der wir für die Implementierung der Methoden genauere Informationen über `Bank` benötigen, den Import durchführen. Da `.m`-Dateien niemals importiert werden, würden somit garantiert keine Zyklen mehr existieren. Um in der `.h`-Datei ein solches Versprechen zu geben, kann das Schlüsselwort `@class` wie folgt eingesetzt werden:

```
@class Bank;
@interface BankAccount : NSObject {
    Bank* bank;
    //...
```

Listing 1.10: Bereitstellen der Klasse `Bank` ohne `#import`

Hiermit ist für die Definition der Instanzvariablen `bank` bekannt, dass es eine Klasse `Bank` gibt, und es gibt keinen Übersetzungsfehler. Wenn nun in der Implementierung der Klasse `BankAccount` weitere Informationen über die Klasse `Bank` benötigt werden, kann der Import der Klassendatei in der Datei `BankAccount.m` durchgeführt werden.

Für die korrekte Übersetzung der Klasse `BankAccount` aus Listing 1.5 benötigt man die beiden Klassen `NSString` und `NSObject`. Da diese beiden Klassen Teil des Cocoa-Frameworks sind, können sie wie folgt importiert werden:

```
#import <Cocoa/Cocoa.h>
```

1.4.3 Vererbung

Beim Programmieren von Klassen treten oft Situationen auf, in denen mehrere Klassen zum Teil äquivalentes Verhalten zeigen. Diese Teile müssten in allen Klassen einzeln implementiert werden. Es gibt also wahrscheinlich gleiche Methoden mit gleicher Implementierung in verschiedenen Klassen. Genau an dieser Stelle setzt die Vererbung an. Sie bietet die Möglichkeit, gleiches Verhalten und damit Quellcode für verschiedene Klassen wiederzuverwenden. Dazu kann für mehrere

Klassen eine **Oberklasse** programmiert werden, in der das Verhalten implementiert ist, das in allen **abgeleiteten Klassen** (auch **Unterklassen** genannt) gleich sein soll. Dabei werden in den Unterklassen sowohl Instanzvariablen als auch Methoden von der Oberklasse geerbt. Eine Klasse, die als Oberklasse verwendet wird, kann ebenfalls eine Oberklasse besitzen. Die sich aus der Vererbung ergebende Hierarchie kann beliebig tief werden. Eine Klasse, die selbst keine Oberklasse besitzt, wird als **Wurzelklasse** bezeichnet, da sie sich in der Vererbungshierarchie auf oberster Stufe befindet. Eine Mehrfachvererbung, wie man sie auch aus manchen objektorientierten Programmiersprachen kennt, bei der eine Klasse mehrere Oberklassen besitzen kann und damit deren Verhalten vereint, gibt es in Objective-C nicht. In Objective-C kann eine Klasse maximal eine Oberklasse besitzen.

Die Wurzelklasse »NSObject«

Unter Mac OS X ist im Cocoa-Framework die Wurzelklasse `NSObject` für Objective-C definiert. Von dieser Klasse leiten sich im Prinzip alle Klassen des Cocoa-Frameworks ab. Ein Teil der Klassenhierarchie, die für das Abbilden von Collections gedacht sind, sehen Sie in Abbildung 1.3. Auch alle Klassen, die man als Programmierer im Zusammenhang mit dem Cocoa-Framework unter Mac OS X implementiert, sollten direkt oder indirekt von `NSObject` erben.

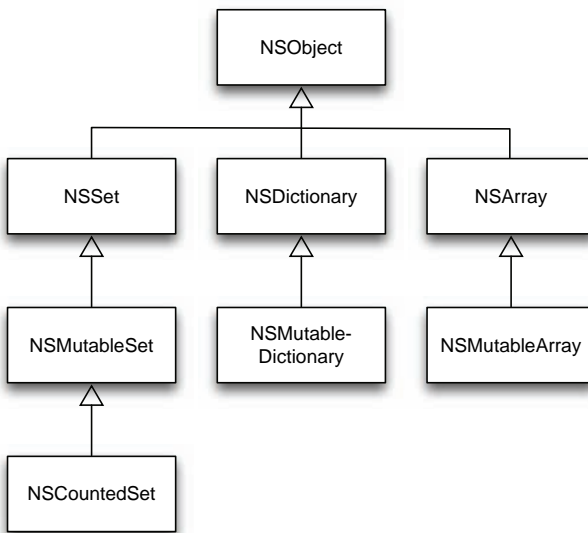


Abb. 1.3: Klassen-Ausschnitt aus dem Cocoa-Framework

In der Klasse `NSObject` ist das Grundverhalten eines Cocoa-Objekts festgelegt. So stellt `NSObject` die Instanzvariable `i sa` bereit. Diese Instanzvariable zeigt bei

jedem erzeugten Objekt auf sein passendes Klassen-Objekt. Auf Basis dieser Instanzvariablen wird das Dynamic Typing und Dynamic Binding betrieben. Hierzu mehr in Kapitel 2.

Um von einer Klasse neue Instanzen erzeugen zu können, sind in der Klasse `NSObject` Methoden für die Objekterzeugung bereitgestellt. Bereits vorgestellt haben wir die Klassenmethode `new`. In Abschnitt 1.5 auf Seite 66 zeigen wir, wie die Objekterzeugung auf Basis von `NSObject` genau funktioniert.

Als Gegenstück zur Objekterzeugung sind in `NSObject` auch die Mechanismen für das Memory Management und damit die Erkennung des Lebensendes eines Objekts implementiert. Als zwei verschiedene Verfahren werden im Cocoa-Framework das Reference Counting und das Garbage Collecting bereitgestellt. Was sich hinter diesen Begriffen verbirgt und was Memory Management wirklich ist, erklären wir in Kapitel 9.

Weiter sehr interessant sind zum Beispiel die Methoden `isMemberOfClass:` und `isKindOfClass:`. Hiermit kann für ein Objekt geprüft werden, ob es exakt eine Instanz einer Klasse ist oder Instanz einer Klasse, die von der als Argument übergebenen Klasse erbt. Diese Funktionalität nennt man **Introspektion**. Die genauen Möglichkeiten der Klasse `NSObject` bzgl. Introspektion betrachten wir in Kapitel 13.

In `NSObject` sind aber noch eine ganze Menge weiterer Methoden bereitgestellt, die einem als Programmierer helfen und durch Vererbung allen Objekten aller abgeleiteten Klassen zur Verfügung stehen. Viele dieser Methoden werden in den einzelnen Kapiteln dieses Buches beschrieben. Es lohnt sich aber in jedem Fall, sich in der Klassendokumentation einen Überblick zu verschaffen.

Erweiterung der Klasse »BankAccount«

Als Beispiel zur Vererbung wollen wir uns ein Konto mit zusätzlichem Überziehungskredit ansehen. Hierzu erbt die Klasse `OverdraftAccount` von der Klasse `BankAccount` aus Listing 1.5:

```
#import <Cocoa/Cocoa.h>
#import "BankAccount.h"

@interface OverdraftAccount : BankAccount {
    float overdraft;
}
- (float) overdraft;
- (void) setOverdraft: (float) overdraft;
- (float) withdraw: (float) money;
@end
```

Listing 1.11: Klasseninterface der Klasse `OverdraftAccount`

Wir wollen uns genauer damit beschäftigen, wie die Instanzvariablen und Methoden vererbt werden. Bei den Instanzvariablen ist die Vererbung eigentlich ganz einfach. Die Instanzvariablen einer Klasse sind nichts anderes als die Summe aller in der Klasse und ihrer Oberklassen definierten Instanzvariablen. Dazu dürfen jedoch keine zwei Instanzvariablen mit gleichem Namen existieren. Andernfalls führt dies zu einem Übersetzungsfehler. Den Instanzen der Klasse `OverdraftAccount` steht also sowohl die Instanzvariable `overdraft` als auch `name`, `accountNumber` und `balance` aus `BankAccount` zur Verfügung. Hierzu kommen noch die von `NSObject` geerbten Instanzvariablen wie `isa` hinzu.

Die Vererbung von Methoden verhält sich etwas komplizierter. Die Methoden, die eine Klasse zur Verfügung stellt, sind ebenfalls alle Methoden, die in der Klasse selbst oder einer ihrer Oberklassen definiert sind. Es werden sowohl Klassenmethoden als auch Methoden für die Instanzen vererbt. Allerdings können hier in den Unterklassen sowohl neue Methoden mit neuen Namen hinzugefügt werden als auch Methoden mit gleichem Namen aus einer Oberklasse überschrieben werden. Gültig ist für eine Klasse immer die erste Methode mit einem Namen, die auf dem Weg über die Oberklasse zur Wurzelklasse hin erreicht wird. Hierdurch kann man als Programmierer die Implementierung einer Methode in den Unterklassen austauschen und anpassen. Bei der Klasse `OverdraftAccount` werden die Methoden `overdraft` und `setOverdraft`: zu den Methoden von `BankAccount` hinzugefügt. Die Methode `withdraw`: wird überschrieben. Eine Implementierung der Methode könnte wie folgt aussehen:

```
- (float) withdraw: (float) money {
    if (money > (balance + overdraft) || money < 0) return balance;
    balance -= money;
    return balance;
}
```

Listing 1.12: Implementierung der Methode `withdraw`:

1.4.4 Instanzvariablen

Wie wir bereits im Listing 1.5 gezeigt haben, erfolgt die Definition der Instanzvariablen im Klasseninterface in geschweiften Klammern direkt hinter dem Namen der Klasse und der Angabe der Superklasse. Die Klasse `BankAccount` besitzt die Instanzvariablen `name`, `accountNumber` und `balance`:

```
@interface BankAccount : NSObject {
    int accountNumber;
    NSString* name;
    float balance;
}
```

Listing 1.13: Instanzvariablen der Klasse `BankAccount`

Außerdem erbt die Klasse `BankAccount` noch die Instanzvariablen der Klasse `NSObject`. Wir wollen bis auf Weiteres davon ausgehen, dies wäre genau die Instanzvariable `isa`.

Per Konvention werden auch Instanzvariablen in Camel Case formuliert (siehe Abschnitt 1.4 auf Seite 44), jedoch beginnen Instanzvariablen mit einem kleinen Buchstaben.

Die Instanzvariablen bilden den internen Zustand einer Instanz der Klasse. Da eine Klasse selbst auch wieder ein Objekt ist, könnte man auf die Idee kommen, dass es etwas Ähnliches wie Instanzvariablen auch für Klassen-Objekte gibt, also Klassenvariablen. Ein solches Konstrukt gibt es in Objective-C allerdings nicht. Wir werden in Abschnitt 1.4.5 auf Seite 60 allerdings zeigen, wie man etwas Vergleichbares mit statischen Variablen erhält.

Wie wir bereits zuvor festgestellt haben, stellt jedes Objekt über die Menge der Methode, die es versteht, eine Abstraktionsbarriere für seine Instanzvariablen dar. Zugegriffen werden kann auf diese Variablen nur aus Methodenimplementierungen der Klasse selbst oder einer ihrer Unterklassen. Die Inhalte der Variablen können also von außerhalb nicht direkt ausgelesen oder geändert werden.

Der Gültigkeitsbereich von Instanzvariablen kann aber noch feingranularer beeinflusst werden. Hierfür gibt es die drei Schlüsselworte `@private`, `@protected` und `@public`. Jedes dieser drei Schlüsselworte stellt eine Art von Gültigkeitsbereich dar. Sie können beliebig einer Menge von Instanzvariablen vorangestellt werden. Für alle darauffolgenden Variablen gilt dann dieser Gültigkeitsbereich, bis ein neuer Gültigkeitsbereich angegeben wird. Standardmäßig wird davon ausgegangen, dass allen Variablen das Schlüsselwort `@protected` vorangestellt ist.

Möchte man hingegen, dass die Variablen `name` und `balance` als `@public` definiert sind, so muss die Definition zum Beispiel wie folgt aussehen:

```
@interface BankAccount : NSObject {
    int accountNumber;

    @public
    NSString* name;
    float balance;
}
```

Listing 1.14: Definition von Instanzvariablen mit `@public`

Die Instanzvariable `accountNumber` bleibt damit weiterhin als `@protected` definiert. Hierdurch können die Instanzvariablen nicht nur aus Methodenimplementierungen der Klasse oder aus Unterklassen heraus zugegriffen werden, sondern auch von außerhalb. So können wir zum Beispiel wie folgt auf die Instanzvariable `name` zugreifen:

```
BankAccount *account = [BankAccount accountWithNumber: 12345678  
                        andName: @"Max Mustermann"];  
NSLog(@"name = %@", account->name);
```

Die Funktion `NSLog` stammt aus dem Cocoa-Framework und dient ähnlich zu `printf` aus C zur Ausgabe von formatierten Zeichenketten auf der Konsole.

Auf eine Variable, die als `@protected` definiert ist, kann auf diese Weise nicht zugegriffen werden. Bisher liefert der Compiler allerdings nur eine Warnung mit dem Hinweis, dass ein Zugriff auf nicht-`@public` Instanzvariablen in Zukunft zu einem echten Fehler führen wird.

Das Problem ist, dass damit die Instanzvariable `name` auch von außen geändert werden kann:

```
account->name = @"Tessa Test";
```

Aus softwaretechnischer Sicht ist es aufgrund des Prinzips des Information Hiding sinnvoll, Instanzvariablen nie als `@public` freizugeben. Besser ist es, für den Zugriff Methoden zur Verfügung zu stellen. Aus diesem Grund sollten Instanzvariablen nur dann als `@public` definiert sein, wenn der Performance-Vorteil durch den direkten Zugriff auf die Speicheradresse ohne zusätzlichen Methodenaufruf wichtig ist. Andernfalls sollte die Instanzvariable mindestens als `@protected` definiert sein.

Vorsicht

Auch die Instanzvariable `isa` der Klasse `NSObject` ist als `@public` definiert. Allerdings sollte man nie auf die Idee kommen, den Inhalt von `isa` zu ändern, da die Variable als Grundlage für das Dynamic Typing und Dynamic Binding verwendet wird (siehe Kapitel 2).

Alternativ kann auch `@private` als Einschränkung des Gültigkeitsbereichs sinnvoll sein. Hierdurch ist der Zugriff nur noch aus Methodenimplementierungen der Klasse möglich. Auch aus Unterklassen heraus ist der Zugriff nicht mehr möglich. Dadurch kann man verhindern, dass eine Unterklasse den Inhalt dieser Instanzvariablen verändern kann. Auch die Unterklasse muss dann die Methoden der Klasse selbst zum Zugriff auf die Variablen verwenden. Für die Instanzvariable `accountNumber` ist es zum Beispiel eher sinnvoll, sie als `@private` zu definieren, denn die Kontonummer eines Kontos darf nicht geändert werden:

```
@interface BankAccount : NSObject {  
    float balance;  
    NSString* name;
```

```
@private
    int accountNumber;
}
```

Listing 1.15: Definition von Instanzvariablen mit `@private`

Als Programmierer der Klasse `BankAccount` kann man dieses verhindern. So wird in der Klasse `BankAccount` aus Listing 1.5 keine Methode `setAccountNumber:` zum Ändern angeboten. Durch einfaches Ableiten der Klasse zu zum Beispiel `MyAccount` wäre dies ohne die Anpassung der Variablen `accountNumber` zu `@private` allerdings möglich:

```
@interface MyAccount : BankAccount {}
- (void) setAccountNumber: (int) value;
@end
```

Die Methode `setAccountNumber:` müsste nur wie folgt implementiert werden:

```
- (void) setAccountNumber: (int) value {
    accountNumber = value;
}
```

Auf einer Instanz von `MyAccount` könnte die Kontonummer dann geändert werden:

```
MyAccount* account = [MyAccount accountWithNumber:1234567 andName:@"Ich"];
[account setAccountNumber:5555555];
```

Da die Variable `accountNumber` oben allerdings als `@private` definiert ist, führt die Implementierung von `setAccountNumber:` zu einem Compiler-Fehler. Der Zugriff auf die Variable `accountNumber` aus einer Unterklasse heraus wird vom Compiler also verhindert.

Wir sehen, dass wir mit den drei Schlüsselwörtern `@private`, `@protected` und `@public` den Gültigkeitsbereich unser Instanzvariablen an die entsprechenden Bedürfnisse anpassen können. In Abbildung 1.4 sehen Sie noch mal eine Übersicht über die Bedeutung der drei Arten der Gültigkeitsbereiche.

Hinweis

Für 64-Bit-Programme kann der Gültigkeitsbereich auch noch auf `@package` eingeschränkt werden. Innerhalb des Kompilats (zum Beispiel ein Framework) ist eine Variable, die mit `@package` definiert ist, wie eine Variable mit `@public` gültig. Außerhalb des Kompilats hingegen verhält sie sich wie eine Variable mit `@private`.

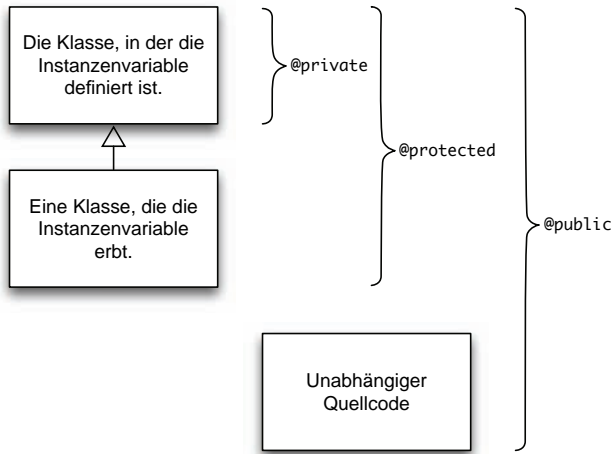


Abb. 1.4: Gültigkeitsbereiche von Instanzvariablen

Speicheraufbau

Nachdem wir uns nun damit beschäftigt haben, wie wir die Instanzvariablen definieren können, wollen wir uns nun der Frage zuwenden, wie ein Objekt auf Basis der Instanzvariablen im Speicher abgebildet wird. In vorherigen Abschnitten haben Sie gelesen, dass die Instanzvariablen den internen Zustand eines Objekts bilden. Aus diesem Grund muss für jede Instanz einer Klasse für jede Instanzvariable ein Platz im Hauptspeicher reserviert werden. Am einfachsten kann man sich vorstellen, dass aus jeder Klasse eine C-Struktur wird, die genau die Instanzvariablen der Klasse enthält. Dann stellt sich allerdings die Frage, was passiert, wenn eine Klasse von einer anderen Klasse erbt. In der Struktur werden dazu auch alle Instanzvariablen, die von anderen Klassen geerbt werden, in der Reihenfolge der Vererbung beginnend bei der Wurzelklasse mit aufgenommen. Sehen wir uns die Struktur an, die die Klasse `BankAccount` aus Listing 1.5 beschreibt:

```
struct BankAccount_structure {  
    Class isa;  
    NSString* name;  
    int accountNumber;  
    float balance;  
};
```

Listing 1.16: Speicherstruktur der Klasse `BankAccount`

Als erstes Element der Struktur steht die Instanzvariable `isa`, die in `BankAccount` von `NSObject` geerbt wurde. Hiernach würden noch weitere Instanzvari-

ablen von `NSObject` folgen. Wir gehen allerdings aus Gründen der Übersichtlichkeit davon aus, dass `NSObject` nur die Instanzvariable `isa` besitzt. Danach folgen dann die Instanzvariablen von `BankAccount` in der im Klasseninterface definierten Reihenfolge. Wird nun eine neue Instanz eines Objekts erzeugt, wird im Hauptspeicher Speicher für die passende Struktur reserviert. Als Objekt wird dann der Zeiger auf diese Struktur zurückgegeben.

Wir wollen uns nun die passende Struktur für die Klasse `OverdraftAccount` aus Listing 1.11 ansehen:

```
struct OverdraftAccount_structure {
    Class isa;
    NSString* name;
    int accountNumber;
    float balance;
    float overdraft;
};
```

Listing 1.17: Speicherstruktur der Klasse `OverdraftAccount`

Wir sehen, dass in dieser Struktur alle Instanzvariablen bis einschließlich `balance` identisch sind, da die Klasse `OverdraftAccount` diese Instanzvariablen von `BankAccount` erbt. Danach folgt dann die Instanzvariable `overdraft` der Klasse `OverdraftAccount`.

Der Vorteil an dieser Reihenfolge ist, dass die Instanzvariablen von `BankAccount` oder `NSObject` auch in einer Struktur für `OverdraftAccount` an der gleichen Position stehen. Der Compiler kann aus diesem Grund den Zugriff auf eine Instanzvariable im Speicher unabhängig von den tatsächlichen Unterklassen berechnen. Die Instanzvariable `isa` befindet sich zum Beispiel immer an der ersten Stelle in der Struktur. Dies ist nicht nur bei Klassen, die von `NSObject` erben, so, sondern bei allen denkbaren Wurzelklassen, da das Objective-C-Laufzeitsystem von allen Wurzelklassen die Bereitstellung der Instanzvariablen `isa` fordert. Im Laufzeitsystem gibt es hierfür die Struktur `objc_object`, die in der Datei `objc/objc.h` definiert ist. Diese Struktur legt auch genau den Datentyp `id` fest:

```
typedef struct objc_object {
    Class isa;
} *id;
```

Listing 1.18: Speicherstruktur eines beliebigen Objective-C-Objekts

Zu diesem Datentyp müssen alle Objekte kompatibel sein.

1.4.5 Methoden und Klassenmethoden

Nachdem wir nun bereits wissen, wie wir eine Klasse aufbauen und wie wir sie verwenden können, wollen wir uns noch einmal etwas genauer damit beschäftigen, wie wir Instanzen- oder Klassenmethoden im Klasseninterface definieren und wie diese Methoden dann implementiert werden können.

Die Definition der Methoden erfolgt im Klasseninterface hinter der Definition der Instanzvariablen. Schauen wir uns dazu noch einmal die Methodendefinitionen der Klasse `BankAccount` aus Listing 1.5 an:

```
@interface BankAccount : NSObject { /*... */}
- (id) initWithNumber: (int) number andName: (NSString*) name;
+ (id) accountWithNumber: (int) number andName: (NSString*) name;

- (int) accountNumber;
- (NSString*) name;
- (float) deposit: (float) money;
- (float) withdraw: (float) money;
- (float) balance;
@end
```

Listing 1.19: Methode der Klasse `BankAccount`

Alle Methodendefinitionen beginnen mit einem Plus (+) oder Minus (-). Das Plus (+) kennzeichnet eine Methode als Klassenmethode, wird also von dem Klassen-Objekt verstanden und das Minus (-) kennzeichnet eine Instanzenmethode. Danach folgt in Klammern der Typ des Rückgabewertes. Zwischen die einzelnen Teile der Methodennamen werden die Typen in Klammern und Variablen der Argumente eingeschoben. Abgeschlossen wird eine Methodendefinition mit einem Semikolon.

Für die Namenskonventionen der einzelnen Teile der Methodennamen bietet sich ebenfalls Camel Case an, wobei die Teile zur Abgrenzung von Klassen mit einem kleinen Buchstaben beginnen sollten. Handelt es sich um einen Teil, der ein zu übergebendes Argument benennt, so muss als Abschluss ein Doppelpunkt folgen. Beim Doppelpunkt zur Einleitung von Argumenten handelt es sich allerdings nicht um eine Konvention, sondern er gehört zur Syntax der Sprache und kann damit auch nicht entfallen.

In vielen Programmen sieht man sehr häufig, dass bei der Definition einer Methode kein Typ für den Rückgabewert spezifiziert ist. Dies liegt daran, dass der Rückgabetyt einer Methode auch entfallen kann. In diesem Fall wird der Compiler annehmen, dass der Rückgabewert dem Typ `id` entspricht. Auf die Methoden `initWithNumber:andName:` und `accountWithNumber:andName:` hätte das Weglassen des Rückgabetyps also keine Auswirkungen. Die verkürzten Definitionen würden also wie folgt aussehen:

```
- initWithNumber: (int) number andName: (NSString*) name;  
+ accountWithNumber: (int) number andName: (NSString*) name;
```

Die eigentliche Implementierung der Methoden erfolgt dann im `@implementation`-Teil in der entsprechenden `.m`-Datei. In diesem Fall zum Beispiel in der Datei `BankAccount.m`:

```
@implementation BankAccount  
// Methodenimplementierungen  
@end
```

Die Implementierung für die Methode `accountNumber` kann zum Beispiel wie folgt aussehen:

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

Eine Methodenimplementierung beginnt also exakt wie die Methodendefinition. Mit dem Unterschied, dass anstelle des Semikolons ein Methodenrumpf in geschweiften Klammern angegeben wird. In der Implementierung von `accountNumber` wird nur der Wert der Instanzvariablen `accountNumber` zurückgegeben. Eine Objective-C-Methode wird immer im Kontext eines Objekts ausgeführt. Dieses Objekt ist in einer versteckten Variablen `self` abgespeichert. Den Zugriff auf die Variable `accountNumber` können wir uns also eher wie folgt vorstellen:

```
- (int) accountNumber {  
    return self->accountNumber;  
}
```

Sehen wir uns die Verwendung dieser Methode in folgendem Beispiel an:

```
BankAccount* account = [BankAccount accountWithNumber:12345678  
                        andName:@"Max Mustermann"];  
NSLog(@"Kontonummer: %d", [account accountNumber]);
```

In diesem Beispiel wird an das Objekt `account` die Nachricht `accountNumber` geschickt. Hierdurch wird obige Implementierung ausgeführt. Die Variable `self` innerhalb der Implementierung zeigt dadurch auf das gleiche Objekt wie die Variable `account`. Von diesem Objekt wird dann die Kontonummer `accountNumber` zurückgegeben und mit `NSLog` auf der Konsole ausgegeben. Das Ergebnis ist die Nummer 12345678, wie man auch in Abbildung 1.5 sieht.

Meldung

Kontonummer: 12345678

Abb. 1.5: Ausgabe der Kontonummer

Innerhalb der Methodenrumpfe darf beliebiger Objective-C-Code verwendet werden. Schauen wir uns dazu als weiteres Beispiel die Implementierung von `deposit`: zum Einzahlen von Beträgen auf das Konto an:

```
- (float) deposit: (float) money {  
    if (money < 0) return balance;  
    balance += money;  
    return balance;  
}
```

Listing 1.20: Implementierung der Methode `deposit`:

Dazu wird als Erstes geprüft, ob der Betrag `money`, der eingezahlt werden soll, negativ ist. In diesem Fall wird keine Einzahlung durchgeführt und der alte Kontostand zurückgegeben. Andernfalls wird auf die Instanzvariable `balance` für den Kontostand der Betrag `money` addiert und der neue Betrag zurückgeliefert.

Wir wollen uns nun noch etwas genauer mit der Variablen `self` und einer zusätzlichen Variablen `super` auseinandersetzen. Erweitern wir dazu die Klasse `BankAccount` um die Methode `saldo`, die angibt, wie viel von einem Konto abgebucht werden kann:

```
- (float) saldo {  
    return [self balance];  
}
```

Listing 1.21: Implementierung der Methode `saldo`

In unserem sehr einfachen Beispielkonto ist der Betrag, den wir abbuchen dürfen, gleich dem Kontostand des Kontos. Für das Auslesen des Kontostands verwenden wir allerdings nicht die Instanzvariable `balance`, sondern senden uns selbst die Nachricht `balance`. Der Vorteil an dieser Variante ist, dass die Implementierung von `saldo` von der Instanzvariablen `balance` entkoppelt ist. Hierdurch kann die interne Darstellung des Kontostands beliebig geändert werden, solange wieder eine funktionierende `balance`-Methode zur Verfügung gestellt wird (Prinzip des Information Hiding). Die Variable `self` dient also nicht nur dem Auslesen von Instanzvariablen, sondern kann auch für das Senden von Methoden verwendet werden. Auf Basis der Methode `saldo` können wir die Methode `withdraw`: wie folgt implementieren:

```
- (float) withdraw: (float) money {
    if (money > [self saldo] || money < 0) return balance;
    balance -= money;
    return balance;
}
```

Listing 1.22: Implementierung der Methode `withdraw`: mit Hilfe von `saldo`

In dieser Implementierung wird die Methode `saldo` dazu verwendet, um den maximal abzubuchenden Betrag zu begrenzen. Auch hierfür wird die Methode dem Empfängerobjekt `self` geschickt.

Da es sich bei `self` um eine normale Variable handelt, könnten wir den Inhalt von `self` auch ändern. Dies hat auch keinen Einfluss auf die weitere Ausführung der Methode. Wir werden diese Möglichkeit bei den Initialisierungsmethoden in Abschnitt 1.5.1 auf Seite 68 noch nutzen. Jedoch ist die Verwendung der Variablen `self`, um ihr einen neuen Wert zuzuweisen, mit Vorsicht zu genießen, da dies schnell zu sehr unverständlichen Programmen führt.

Wir wollen nun untersuchen, was unter der Variablen `super` zu verstehen ist. In der Klasse `OverdraftAccount` aus Listing 1.11 kann nun die Methode `saldo` überschrieben werden:

```
- (float) saldo {
    return [self overdraft] + [super saldo];
}
```

Listing 1.23: Überschreiben der Methode `saldo` in der Klasse `OverdraftAccount`

Die Variable `super` stellt eine Sonderrolle dar. Es handelt sich eigentlich um die Variable `self`, jedoch mit dem Hinweis, dass bei der Methodenauflösung die Implementierung in der Superklasse der aktuellen Klasse gesucht werden soll. Die Berechnung des Saldos in der Klasse `OverdraftAccount` ergibt sich also aus dem Überziehungskredit (`[self overdraft]`) und aus dem Saldo, der sich aus der Implementierung in der Superklasse `BankAccount` ergibt (`[super saldo]`). Da es sich bei der Variablen `super` mehr um eine Compilerdirektive für die Variable `self` als um eine »echte« Variable handelt, kann man der Variablen `super` keinen neuen Wert zuweisen.

Durch das Überschreiben der Methode `saldo` ändert sich indirekt auch das Verhalten der Methode `withdraw`: , ohne dass wir diese Methode in der Klasse `OverdraftAccount` wie in den vorigen Abschnitten gezeigt überschreiben. Verwendet man die Methode `withdraw`: mit einer Instanz der Klasse `BankAccount`, wird die alte nicht überschriebene Variante von `saldo`, die direkt den Kontostand zurückgibt, verwendet, um zu prüfen, ob der Betrag abgebucht werden kann. Wird allerdings eine Instanz vom Typ `OverdraftAccount` verwendet, ruft die Methode

`withdraw`: die neue Version von `saldo` auf, wodurch sich der abzubuchende Betrag um den Überziehungskredit `overdraft` erhöht. Hierzu folgendes Beispiel:

```
BankAccount* account;
account = [BankAccount accountWithNumber:1 andName:@" 1"];
NSLog(@"Kontostand: %.2f", [account withdraw: 300]);

account = [OverdraftAccount accountWithNumber:2
           andName:@"2" andOverdraft:300];
NSLog(@"Kontostand: %.2f", [account withdraw: 300]);
```

Listing 1.24: Unterschiedliches Verhalten der Methode `withdraw`:

In der Ausgabe in Abbildung 1.6 sieht man, dass bei einem Konto vom Typ `BankAccount` der Abhebevorgang nicht durchgeführt wird, wohingegen das Abheben beim Konto vom Typ `OverdraftAccount` funktioniert.

```
Meldung
Kontostand: 0.00
Kontostand: -300.00
```

Abb. 1.6: Abheben mit überschriebener `saldo`-Methode

Es gibt für Objekte jedoch noch eine weitere Möglichkeit, auf Methoden zu reagieren, die nicht über den Klassenmechanismus implementiert sind. So ist es möglich, dass unterschiedliche Objekte ein und derselben Klassen unterschiedlich reagieren. Das Prinzip dieser Methodenauswertung nennt man **Forward Invocation** und wird in Abschnitt 2.4 beschrieben.

Klassenmethoden und statische Variablen

Kommen wir nun noch einmal auf die Definition und Implementierung von Klassenmethoden zurück. Definieren wir dazu in der Klasse `BankAccount` die Klassenmethode `bankBalance`, die die Summe der Kontostände aller `BankAccount`-Objekte darstellen soll:

```
+ (float) bankBalance;
```

Mit diesen Methoden können wir dann den Kontostand über alle Bankkonten hinweg auslesen:

```
BankAccount *account;
account = [BankAccount accountWithNumber:1 andName:@"Konto 1"];
[account deposit:100];
account = [BankAccount accountWithNumber:2 andName:@"Konto 2"];
```

```
[account deposit:200];
account = [BankAccount accountWithNumber:3 andName:@"Konto 3"];
[account deposit:150];
[account withdraw:100];

NSLog(@"Kontostandsumme: %.2f", [BankAccount bankBalance]);
```

Listing 1.25: Verwendung der Klassenmethode `bankBalance`

Wir wollen dazu die Implementierung der Klasse `BankAccount` so anpassen, dass sich die Ausgabe in Abbildung 1.7 ergibt.

Meldung

```
Kontostandsumme: 350.00
```

Abb. 1.7: Summe der Kontostände mit der Methode `bankBalance`

Um diese Funktionalität zu implementieren, werden eigentlich Klassenvariablen benötigt, die es in Objective-C allerdings nicht gibt. Da jedes Klassen-Objekt jedoch nur einmal existiert, kann man als Alternative statische Variablen definieren. Hierzu muss der Variablendefinition nur das Schlüsselwort `static` vorangestellt werden. Wir definieren uns dazu eine statische Variable `bankBalance` im Implementationsblock, das heißt, die Variable ist nur innerhalb der Klassenimplementierung sichtbar:

```
@implementation BankAccount

static float bankBalance = 0.0;
//Methodenimplementierungen
@end
```

Listing 1.26: Statische Variablen als Ersatz für Klassenvariablen

Der Inhalt der Variablen dient nun als Rückgabewert für die Methode `bankBalance`:

```
+ (float) bankBalance {
    return bankBalance;
}
```

Listing 1.27: Implementierung der Klassenmethode `bankBalance`

Da es sich bei `bankBalance` nicht um eine Klassenvariable, sondern um eine statische Variable handelt, kann der Zugriff darauf nicht durch `self->bankBalance` ersetzt werden, obwohl `self` auf das Klassen-Objekt zeigt.

Hinweis

Statische Variablen können auch in anderer Umgebung verwendet werden. So kann man zum Beispiel innerhalb einer Methode eine statische Variable definieren, die nur innerhalb dieser Methode sichtbar, aber über die Ausführung der Funktion hinaus gültig ist.

In den Implementierungen von `withdraw:` und `deposit:` muss jede Bankbewegung nun noch in der Variablen `bankAccount` vermerkt werden:

```
- (float) deposit: (float) money {
    if (money < 0) return balance;
    balance += money;
    bankBalance += money;
    return balance;
}
- (float) withdraw: (float) money {
    if (money > [self saldo] || money < 0) return balance;
    balance -= money;
    bankBalance -= money;
    return balance;
}
```

Listing 1.28: Anpassen der Methoden `deposit:` und `withdraw:`

Natürlich könnte man in einer Klassenmethode ebenfalls die Variable `self` für das Senden von Methoden an das Klassen-Objekt verwenden. Experimente hiermit überlassen wir allerdings dem Leser, wir kommen bei der Implementierung von Konstruktoren in Abschnitt 1.5.2 auf Seite 71 darauf zurück.

1.5 Erzeugung von Objekten

Wie erzeugen wir als Programmierer nun ein Objekt, sprich eine neue Instanz einer Klasse? Im Gegensatz zu vielen anderen objektorientierten Programmiersprachen ist die Objekterzeugung bei Objective-C nicht in die Syntax der Programmiersprachen fest eingebaut, sondern ist Bestandteil des Frameworks. Im Cocoa-Framework ist die Objekterzeugung in der Klasse `NSObject` über eine Kombination aus der Methode `alloc` für die **Allokation** von Speicher für die Instanz und einer passenden `init`-Methode für die **Initialisierung** der erzeugten Instanz gelöst. Eine Instanz der Klasse `NSObject` oder `BankAccount` aus Listing 1.5 erhalten wir durch die folgenden Programmzeilen:

```
NSObject *obj = [[NSObject alloc] init];
BankAccount *account = [[BankAccount alloc]
    initWithNumber: 12345678 andName: @"Max Mustermann"];
```

Die `alloc`-Methode ist eine Klassenmethode der Klasse `NSObject`. Sie fordert vom Laufzeitsystem für die Speicherung des Objekts ausreichend Speicher an und reserviert ihn. Als Ergebnis liefert die Funktion den Pointer auf diese Adresse als die Instanz selbst zurück. Der Speicherbereich ist außerdem von der Methode `alloc` komplett mit Nullen gefüllt worden, um nicht zufällig auf Daten eines vorherigen Objekts zugreifen zu können. Alternativ gibt es noch die Methode `allocWithZone:`, die ebenfalls den Speicher für ein neues Objekt anfordert. Auf die Funktionsweise dieser Methode wollen wir später im Abschnitt 9.1 eingehen.

Wirklich initialisiert ist das neue Objekt allerdings noch nicht. Dazu dient die Methode `init` oder ähnliche Methoden wie `initWithNumber:andName:` der Klasse `BankAccount`. Es sollten auch nur diese Methoden überschrieben werden, um den Inhalt von Instanzvariablen eines Objekts mit Standardwerten zu initialisieren. Die `alloc`-Methode sollte nach Möglichkeit nie überschrieben werden. Sehen wir uns an, wie wir die Initialisierungsmethode `initWithNumber:andName:` der Klasse `BankAccount` implementieren könnten:

```
- (id) initWithNumber: (int) aNumber andName: (NSString*) aName {
    if (self = [super init]) {
        accountNumber = aNumber;
        name = aName;
    }
    return self;
}
```

Listing 1.29: Implementierung der Initialisierungsmethode `initWithNumber:andName:`

In der Methode `initWithNumber:andName:` wird zuerst die Initialisierung an die `init`-Methode einer der Superklassen weitergereicht. Ist die Initialisierung durch die Superklassen erfolgreich verlaufen, so werden die Instanzvariablen `accountNumber` und `name` mit den Werten der entsprechenden Parameter der Methode belegt. Zum Schluss wird die Instanz der Klasse als Ergebnis zurückgegeben (`self`). Auf den Grund für die `if`-Abfrage werden wir im nächsten Abschnitt eingehen.

Nach diesen beiden Schritten steht im Laufzeitsystem ein neues Objekt, das fertig initialisiert ist, im Speicher bereit. Es handelt sich bei dieser Form der Objekterzeugung allerdings nur um eine Konvention, die durch die Oberklasse `NSObject` und einige zusätzliche Regeln festgelegt ist, und könnte natürlich auch beliebig anders implementiert werden. So kann zum Beispiel die Methode `alloc` auch überschrieben werden. Allerdings empfiehlt Apple ausdrücklich, dies nicht zu tun.

Über die Vorteile und Nachteile einer in Allokation und Initialisierung getrennten Konstruktion von Objekten kann man lange diskutieren. Der Vorteil dieser Vari-

ante gegenüber den Varianten in anderen objektorientierten Programmiersprachen ist allerdings, dass die Objekterzeugung erst in den Klassen selbst (in unserem Fall in der Klasse `NSObject` und von dort vererbt) implementiert und nicht Bestandteil des Laufzeitsystems und der Sprache ist. Hierdurch hat der Programmierer kompletten Einfluss auf die Objekterzeugung und kann, falls er dies will, zum Beispiel eigene Optimierungsmethoden zur Allokation seiner Objekte programmieren.

Es gibt in Objective-C unter dem Cocoa-Framework jedoch in eigentlich jeder Klasse Methoden, die die Objekterzeugung in einem Schritt zusammenfassen und nicht in Allokation und Initialisierung auftrennen. Allerdings handelt es sich hierbei ebenfalls nur um eine Konvention. Diese Konstruktoren werden von Apple auch als **Convenience-Konstruktoren** bezeichnet. Jedoch bedienen sich auch die Convenience-Konstruktoren ebenfalls der `alloc`-Methode und einer Initialisierungsmethode. In Abschnitt 1.5.2 auf Seite 71 werden wir uns mit den Convenience-Konstruktoren noch etwas genauer beschäftigen.

Es gibt in `NSObject` allerdings auch noch die bereits kennen gelernte Methode `new`. Diese Methode verbindet ebenfalls die Allokation und Initialisierung für die Konstruktion eines neuen Objekts. Sie spielt, was das Memory Management angeht, aber eine etwas andere Rolle als die Convenience-Konstruktoren. Implementiert ist die `new`-Methode wie folgt:

```
+ (id) new {  
    return [[self alloc] init];  
}
```

Listing 1.30: Implementierung der Klassenmethode `new` von `NSObject`

Hiermit lässt sich die Konstruktion des obigen Objekts wie folgt erledigen:

```
NSObject *obj = [NSObject new];
```

1.5.1 Initialisierungsmethoden

Als Bestandteil der Objekterzeugung spielen die Initialisierungsmethoden eine sehr wichtige Rolle. Sie dienen dem Programmierer dazu, die Instanzvariablen eines Objekts auf Startwerte festzulegen. Per Konvention beginnen alle Initialisierungsmethoden mit `init`, gefolgt von weiteren Teilen für die Benennung der Argumente. Der Typ der Rückgabe ist bei allen Initialisierungsmethoden `id`. In `NSObject` steht bereits die Initialisierungsmethode `init` ohne Argumente bereit. Diese Methode kann in Unterklassen überschrieben werden oder um weitere Initialisierungsmethoden mit zusätzlichen Argumenten ergänzt werden.

Schauen wir uns zuerst an, wie eine Implementierung für die Methode `init` für die Klasse `BankAccount` aus Listing 1.5 aussehen könnte:

```
- (id) init {
    if (self = [super init]) {
        accountNumber = 1111;
        name = @"Some Name";
    }
    return self;
}
```

Listing 1.31: Implementierung einer Initialisierungsmethode `init`

Wir sehen, dass als Erstes die `init`-Methode der Superklasse, also in diesem Fall die `init`-Methode von `NSObject`, ausgeführt wird. Diese Methode liefert als Ergebnis im Normalfall die Instanz, die gerade initialisiert wird, zurück. Diese wird der Variablen `self` zugewiesen. Falls aus irgendeinem Grund die Objekterzeugung nicht erlaubt ist, so kann eine Initialisierungsmethode `nil` zurückgeben. Um diesen Fall abzufangen, dient die `if`-Abfrage. Falls die Initialisierung der Superklassen erfolgreich war, werden die Instanzvariablen `accountNumber` und `name` auf Standardwerte gesetzt. Zum Schluss wird der Wert der Variablen `self` als Ergebnis zurückgegeben. Hierdurch wird das initialisierte Objekt zurückgegeben, falls die Initialisierung durch die Superklasse erfolgreich war und andernfalls ebenfalls `nil`. Auf diese Art ist eine Initialisierungsmethode eigentlich immer aufgebaut. Der Vorteil an dieser Art ist, dass durch die Initialisierung die Objekterzeugung auch verweigert werden kann, indem `nil` zurückgegeben wird. Allerdings muss in diesem Fall darauf geachtet werden, dass das Objekt zuvor wieder aus dem Speicher entfernt wird.

Da bei einem Bankkonto eigentlich die Instanzvariablen `accountNumber` und `name` initialisiert werden sollten, sollte entsprechend zur Initialisierung die Methode `initWithNumber:andName:` verwendet werden. Die Implementierung dieser Methode ist im Grunde genommen äquivalent zu der `init`-Methode zuvor. Allerdings werden die Werte der Instanzvariablen `accountNumber` und `name` durch die Parameter der Methode bestimmt:

```
- (id) initWithNumber: (int) aNumber andName: (NSString*) aName {
    if (self = [super init]) {
        accountNumber = aNumber;
        name = aName;
    }
    return self;
}
```

Listing 1.32: Implementierung der Initialisierungsmethode `initWithNumber:andName:`

Die Rückgaben von `nil` können wir dann verwenden, um die Objekterzeugung ohne Parameter mit `init` wie folgt zu verweigern:

```
- (id) init {  
    [self release];  
    return nil;  
}
```

Listing 1.33: Verboten der Initialisierungsmethode `init`

Zuerst wird der für das erzeugte Objekt reservierte Speicher mit der Methode `release` wieder freigegeben. Wie genau die Methode funktioniert, erklären wir in Kapitel 9. Danach wird `nil` zurückgegeben, wodurch die Objekterzeugung abgebrochen wird. Auch die Erzeugung eines neuen Objekts mit der Klassenmethode `new` führt damit nicht mehr zu einem neuen Objekt, da auch die Implementierung von `new` auf `init` zurückgreift:

```
BankAccount *account = [BankAccount new];
```

Dies hat zur Folge, dass man als Programmierer mit einer nicht erfolgreichen Objekterzeugung rechnen muss. In dem obigen Beispiel zeigt die Variable `account` nach dem Aufruf von `new` auf `nil`. Aus diesem Grund sollte in einem Fall wie der überschriebenen `init`-Methode auf das Auslösen einer Ausnahme zur Verweigerung der Ausführung nachgedacht werden. Hierdurch führt die Ausführung eines nicht erlaubten Konstruktors wirklich zu einem Fehler und lässt sich damit nicht so leicht übersehen (siehe hierzu Kapitel 6).

Bei der Implementierung von Initialisierungsmethoden kann jedoch auch auf andere Initialisierungsmethoden mit Argumenten zurückgegriffen werden. Wir wollen hierfür eine Initialisierung für die Klasse `OverdraftAccount` aus Listing 1.11 programmieren, bei der zusätzlich zu Kontonummer und Name auch der Überziehungskredit initialisiert wird. Die Methode `initWithNumber:andName:andOverdraft:` ist wie folgt implementiert:

```
- (id) initWithNumber: (int) aNumber  
    andName: (NSString*) aName  
    andOverdraft: (int) aOverdraft {  
  
    if (self = [super initWithNumber: aNumber andName:aName]) {  
        overdraft = aOverdraft;  
    }  
    return self;  
}
```

Listing 1.34: Implementierung von `initWithNumber:andName:andOverdraft:`

Die Initialisierung von `name` und `accountNumber` wird an die Methode `initWithNumber:andName:` weitergereicht. An dieser Stelle könnte auch `self` anstelle von `super` verwendet werden. Durch den Aufruf von `[self initWithNumber:`

`aNumber andName:aName]` kann jedoch nicht sichergestellt werden, dass die erwartete Implementierung von `initWithNumber:andName:` aus der Klasse `BankAccount` ausgeführt wird, wenn die Methode in einer eventuellen Unterklasse überschrieben wird.

Ist diese Initialisierung mit `initWithNumber:andName:` erfolgreich, wird die Instanzvariable `overdraft` auf den Wert des Parameters `aOverdraft` gesetzt.

Initialisierung von Klassen-Objekten

Aber auch Klassen-Objekte besitzen eine Initialisierungsmethode `initialize:`

```
+ (void) initialize {  
    // Klasseninitialisierungen  
}
```

Listing 1.35: Klasseninitialisierung mit der Klassenmethode `initialize`

Diese Methode ist in der Klasse `NSObject` als leere Methode implementiert. Bei der ersten Verwendung einer Klasse wird für diese Klasse die Methode `initialize` ausgeführt. Die Methode kann damit dazu genutzt werden, um statische Variablen (Klassenvariablen) zu initialisieren. Allerdings muss man vor allem beachten, dass, wenn man eine eigene Wurzelklasse programmiert, diese Methode vorhanden sein muss, da es sonst zu einem Laufzeitfehler kommt, wenn das Laufzeitsystem für eine Klasse versucht, diese Methode auszuführen (siehe Abschnitt 9.3.6).

1.5.2 Convenience-Konstruktoren

Die Convenience-Konstruktoren verbinden die Allokation und Initialisierung zu einer einstufigen Objekterzeugung. Für unsere Klasse `BankAccount` aus Listing 1.5 ist die Methode `accountWithNumber:andName:` ein Convenience-Konstruktor. Der Konstruktor ist auf folgende Weise implementiert:

```
+ (id) accountWithNumber: (int) number andName: (NSString*) name {  
    return [[[self alloc] initWithNumber:number andName:name] autorelease];  
}
```

Listing 1.36: Implementierung des Convenience-Konstruktors
`accountWithNumber:andName:`

Zuerst wird an das Objekt in der Variablen `self` die Nachricht `alloc` gesendet. Da die Methode `accountWithNumber:andName:` eine Klassenmethode ist, zeigt `self` auf das Klassen-Objekt, von dem eine Instanz erzeugt werden soll. Diese Instanz wird durch die Methode `alloc` erzeugt. Mit der Methode `initWithNumber:andName:` wird dann diese Instanz initialisiert. Zum Schluss wird auf dieser Instanz noch die Methode `autorelease` aufgerufen. Hierbei handelt es sich um eine

Methode aus der Klasse `NSObject`, die für das Memory Management benötigt wird (siehe Kapitel 9). Dieser Aufruf gehört eigentlich in jeden Convenience-Konstruktor.

Mit dem Konstruktor `accountWithNumber:andName:` kann man dann in einem Schritt eine Instanz der Klasse `BankAccount` erstellen:

```
BankAccount *account = [BankAccount  
    accountWithNumber: 12345678 andName: @"Max Mustermann"];
```

Durch das Senden der Nachricht `alloc` an die Variable `self` kann der Konstruktor allerdings auch direkt für die Konstruktion von Instanzen der Unterklassen von `BankAccount` verwendet werden. Dies sieht zum Beispiel für die Klasse `OverdraftAccount` aus Listing 1.11 wie folgt aus:

```
OverdraftAccount *account = [OverdraftAccount  
    accountWithNumber: 87654321 andName: @"Tessa Test"];
```

Der Konstruktor wurde also für die Klasse `OverdraftAccount` weitervererbt. In der Klasse `OverdraftAccount` kann man natürlich aber auch noch weitere Konstruktoren hinzufügen:

```
+ (id) accountWithNumber: (int) number  
    andName: (NSString*) name  
    andOverdraft: (int) overdraft {  
  
    return [[[self alloc]  
        initWithNumber:number  
        andName:name  
        andOverdraft:overdraft] autorelease];  
}
```

Listing 1.37: Implementierung von `accountWithNumber:andName:andOverdraft:`

Dieser Konstruktor kann verwendet werden, um eine Instanz der Klasse `OverdraftAccount` oder von Unterklassen dieser zu erzeugen, aber nicht für die Klasse `BankAccount`:

```
OverdraftAccount* account = [OverdraftAccount  
    accountWithNumber:2222222 andName:@"Du" andOverdraft:300];
```

Wie bei allem, was die Objekterzeugung angeht, handelt es sich bei Convenience-Konstrukturen wiederum nur um eine Konvention. Es ist allerdings sinnvoll, sie genau wie hier beschrieben einzusetzen. Denn so verstehen auch andere Pro-

grammierer, die unsere Klasse verwenden, die Funktionsweise ihrer Konstruktoren ohne zusätzliche Dokumentation.

Aus diesem Grund sollten auch gewisse Namenskonventionen eingehalten werden. So sollten alle Convenience-Konstruktoren einer Klasse das gleiche Präfix besitzen, das verständlich macht, welche Art von Instanz erzeugt wird. In unserem Fall beginnen alle Konstruktoren mit dem Präfix `account`. Bei der Klasse `NSString` ist dies zum Beispiel `string`. Alle darauffolgenden Teile des Namens sollten sich an einer Initialisierungsmethode orientieren. So gibt es eigentlich für jeden Convenience-Konstruktor auch eine passende Initialisierungsmethode. In unserem Fall also `accountWithNumber:andName:` und `initWithNumber:andName:` oder `accountWithNumber:andName:andOverdraft:` und `initWithNumber:andName:andOverdraft:`.

Es spricht jedoch nichts dagegen, in einem Convenience-Konstruktor auch weitere Initialisierungen vorzunehmen. Der Konstruktor `accountWithNumber:andName:andOverdraft:` könnte zum Beispiel mit dem Konstruktor `accountWithNumber:andName:` und der Methode `setOverdraft:` realisiert werden:

```
+ (id) initWithNumber: (int) number
        andName: (NSString*) name
        andOverdraft: (int) overdraft {

    OverdraftAccount* account =
        [self initWithNumber:number andName:name];
    [account setOverdraft:overdraft];
    return account;
}
```

Listing 1.38: Alternative Implementierung von `initWithNumber:andName:andOverdraft:`

In diesem Fall sollte man allerdings nicht auf die Idee kommen, die Variable `self` für die Instanz wiederzuverwenden, da es bei größeren Methoden zu sehr unverständlichem Quellcode führen würde:

```
self = [self initWithNumber:number andName:name];
[self setOverdraft:overdraft];
return self;
```

Denn hierdurch zeigt bis zur Konstruktion des neuen Objekts die Variable `self` auf das Klassen-Objekt und nach der Konstruktion auf das neue Objekt. Der Unterschied ist jedoch nur schwer zu erkennen und die Methode sollte deshalb lieber wie in der ersten Variante realisiert werden.

1.6 Übungsaufgaben

1. Aufgabe

Schreiben Sie das folgende Programm, bei dem alle Objekt-Variablen dynamisch typisiert sind, in ein äquivalentes Programm um, in dem alle Variablen statisch typisiert sind:

```
id array = [NSMutableArray array];
for (int i = 0; i < 10; i++) {
    id formatStr = @"Bankkunde Nr. %d";
    id name = [[NSString alloc] initWithFormat:formatStr, i];
    id account = [OverdraftAccount
        initWithNumber:0 andName:name andOverdraft: i * 80];
    [array addObject:account];
}
for (int i = 0; i < 10; i++) {
    id account = [array objectAtIndex:(9 - i)];
    float value = i * 78.33;
    if (i % 2 == 0) {
        [account withdraw:value];
    } else {
        [account deposit: value];
    }
}
float sum = [BankAccount bankBalance];
id message = @"Summe aller Kontostände ist %.2F";
NSLog(message, sum);
```

2. Aufgabe

Schreiben Sie ein Klasseninterface für eine Klasse Bank zur Modellierung einer Bank. Eine Bank besitzt einen Namen (name) und eine BLZ (bankCode). Stellen Sie hierfür Instanzenvariablen und Methoden für den Zugriff bereit. Stellen Sie außerdem eine Instanzenvariable accounts vom Typ NSMutableArray* bereit. Weiterhin soll eine Bank Methoden für das Anlegen eines Kontos auf Basis des Kundennamens sowie das Suchen eines Kontos auf Basis der Kontonummer anbieten. Hierfür können Sie die Klasse OverdraftAccount verwenden. Für die Erstellung eines Bank-Objekts sollen sowohl eine Initialisierungsmethode als auch ein Convenience-Konstruktor existieren, denen man sowohl den Namen der Bank als auch die BLZ übergibt.

3. Aufgabe

Implementieren Sie die Initialisierungsmethode und den Convenience-Konstruktor aus Aufgabe 2. Erzeugen Sie dazu ein neues `NSMutableArray`-Objekt mit der Klassenmethode `array` für die Instanzvariable `accounts`.

4. Aufgabe

Bilden Sie von der Klasse `Bank` eine Unterklasse `OnlineBank` mit zusätzlicher Instanzvariablen `url`. In der Instanzvariablen `url` soll die Adresse zur WWW-Seite als `NSString`-Objekt gespeichert sein. Bieten Sie für diese Instanzvariable ebenfalls Zugriffsmethoden an, über die die Instanzvariable `url` ausgelesen und geändert werden kann. Geben Sie für diese Klasse sowohl das Klasseninterface als auch eine C-Struktur für die Speicherstruktur der Klasse `OnlineBank` an. Gehen Sie davon aus, dass die Klasse `NSObject` nur die Instanzvariable `isa` besitzt.