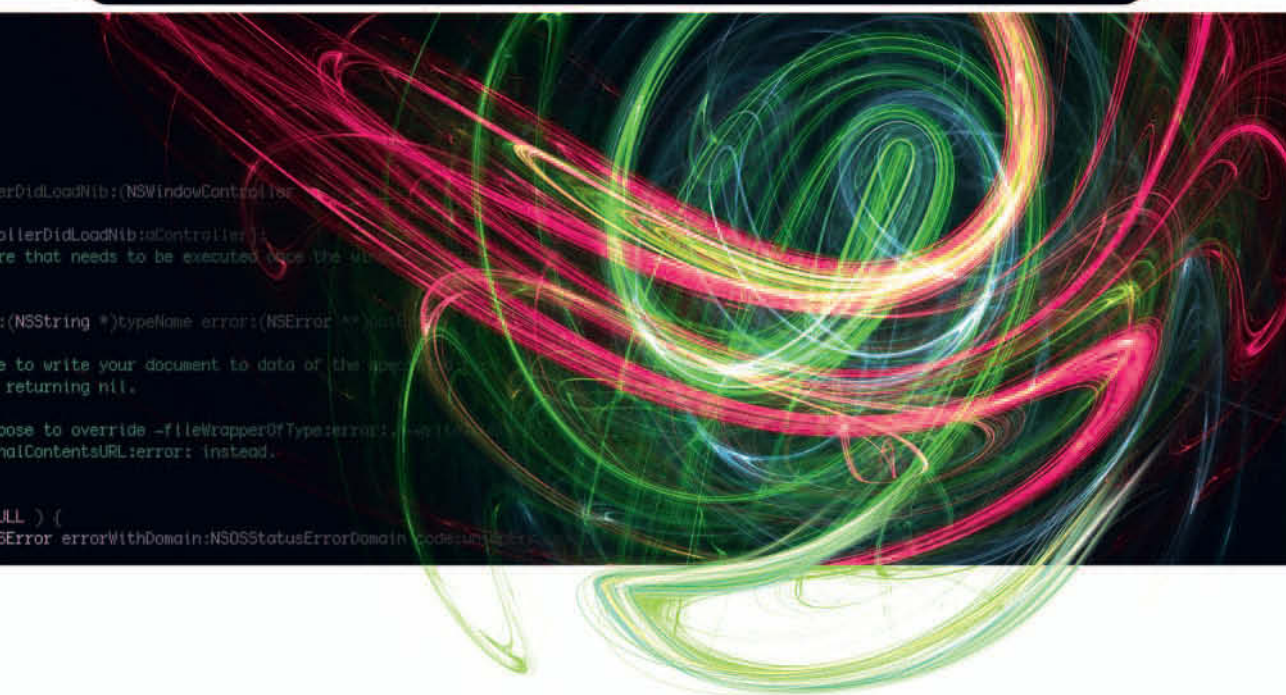
Stephen G. Kochan

Objective-C 2.0

Anwendungen entwickeln für Mac und iPhone



ADDISON-WESLEY

3 Klassen, Objekte und Methoden

In diesem Kapitel werden Sie etwas zu einigen wesentlichen Konzepten in der objektorientierten Programmierung lernen und anfangen, in Objective-C mit Klassen zu arbeiten. Sie müssen ein wenig Terminologie lernen, aber wir werden es recht allgemeinsprachlich halten. Wir betrachten hier nur einige der grundlegenden Begriffe, damit Sie nicht zu viel auf einmal bewältigen müssen. In *Anhang A, Glossar,* am Ende dieses Buchs finden Sie genauere Definitionen.

3.1 Was ist eigentlich ein Objekt?

Ein *Objekt* ist ein Gegenstand. Denken Sie bei der objektorientierten Programmierung an Gegenstände und das, was Sie mit diesen Gegenständen machen möchten. Dies steht im Gegensatz zu einer prozeduralen Programmiersprache wie C. In C überlegen Sie gewöhnlich erst, was Sie tun möchten, und machen sich dann Gedanken über die Objekte, was fast das Gegenteil von Objektorientierung ist.

Nehmen wir ein Beispiel aus dem Alltagsleben. Wir gehen davon aus, dass Sie ein Auto haben. Dies stellt zweifellos einen Gegenstand dar, und zwar einen in Ihrem Besitz. Sie haben nicht einfach irgendein Fahrzeug, sondern ein bestimmtes, das in einer Fabrik produziert wurde, sei es in Wolfsburg, Japan oder wo auch immer. Ihr Wagen hat eine Fahrgestellnummer, die es eindeutig bestimmt.

Im Sprachgebrauch der Objektorientierung stellt Ihr Auto die *Instanz* eines Autos dar. Wenn wir in der Terminologie bleiben, ist Auto der Name der Klasse, von der diese Instanz gebildet worden ist.

Somit wird jedes Mal, wenn ein neuer Wagen vom Band läuft, eine neue Instanz der Klasse der Autos erstellt, und jede Instanz des Autos wird als Objekt bezeichnet.

Ihr Fahrzeug kann beispielsweise in Silber lackiert sein, eine schwarze Innenausstattung haben und ein Cabrio oder eine Limousine sein. Zudem führen Sie mit Ihrem Wagen bestimmte Handlungen aus. So fahren Sie Ihr Auto, betanken es, waschen es (hoffentlich) ab und zu, lassen an ihm eine Inspektion vornehmen usw. Tabelle 3.1 verdeutlicht dies.

Die in Tabelle 3.1 aufgeführten Aktionen können mit Ihrem Wagen ausgeführt werden, aber auch mit anderen Fahrzeugen. So fährt zum Beispiel Ihre Schwester ihr eigenes Auto, wäscht es, betankt es usw.

Objekt	Aktion
Ihr Auto	Fahren
	Betanken
	Waschen
	Inspizieren lassen

▶ *Tabelle 3.1: An Objekten ausgeführte Tätigkeiten*

3.2 Instanzen und Methoden

Eine einzigartige Ausprägung einer Klasse ist eine Instanz, und die Handlungen, die an dieser Instanz ausgeführt werden, nennen sich *Methoden*. In einigen Fällen kann eine Methode auf eine Instanz der Klasse oder auf die Klasse selbst angewendet werden. So bezieht sich zum Beispiel das Waschen Ihres Autos auf eine Instanz (in der Tat können alle in Tabelle 3.1 aufgeführten Methoden als Instanzmethoden betrachtet werden). Herauszufinden, wie viele Wagentypen ein Hersteller produziert, bezieht sich auf die Klasse, sodass dies eine Klassenmethode ist.

Nehmen wir an, Sie haben zwei Autos aus der gleichen Produktionsserie, die identisch zu sein scheinen: Beide haben die gleiche Innenausstattung, die gleiche Lackierung usw. Sie sind zu Beginn vielleicht gleich, aber jedes einzelne nimmt einmalige Eigenschaften an, während es von seinem jeweiligen Besitzer benutzt wird. So könnte das eine im Laufe der Zeit einen Kratzer abbekommen, und das andere mehr Kilometer zurücklegen. Jede Instanz bzw. jedes Objekt enthält nicht nur Informationen zu seinen ursprünglichen Eigenschaften im fabrikneuen Zustand, sondern auch zu seinen aktuellen Merkmalen. Diese können sich im Lauf der Zeit verändern. Während Sie Ihr Auto fahren, leert sich der Benzintank, der Wagen verdreckt und die Reifen fahren sich ab.

Eine Methode auf ein Objekt anzuwenden, kann den *Zustand* des Objekts verändern. Wenn Ihre Methode darin besteht, Ihr »Auto zu betanken«, ist der Benzintank Ihres Wagens gefüllt, wenn die Methode ausgeführt wurde. Sie hat dann den Zustand des Tanks Ihres Autos verändert.

Die wesentlichen Prinzipien, die hier zugrunde liegen, bestehen darin, dass Objekte einzigartige Vertreter einer Klasse sind und dass jedes Objekt bestimmte Informationen (bzw. Daten) enthält, die normalerweise diesem Objekt eigen sind. Die Methoden bilden die Mittel für den Zugriff auf diese Daten und ihre Änderung.

Die Programmiersprache Objective-C weist die folgende besondere Syntax auf, um Methoden auf Klassen und Instanzen anzuwenden:

```
[ KlasseOderInstanz methode ];
```

In dieser Syntax folgt einer öffnenden Klammer der Name einer Klasse oder einer Instanz dieser Klasse, dann ein oder mehrere Leerzeichen gefolgt von der Methode, die Sie ausführen wollen. Die Syntax endet mit einer schließenden Klammer und einem abschließenden Semikolon. Wenn Sie eine Klasse oder eine Instanz dazu auffordern, eine bestimmte Aktion auszuführen, sagt man dazu, dass Sie ihm eine *Nachricht* senden; der Empfänger dieser Nachricht wird *Adressat* genannt. Eine andere Möglichkeit, das beschriebene allgemeine Format zu beschreiben, lautet daher wie folgt:

```
[ Empfänger nachricht ] ;
```

Gehen wir zu der zuvor aufgeführten Liste zurück und schreiben alles in dieser neuen Syntax. Bevor Sie das tun, müssen Sie aber zu Ihrem neuen Auto kommen. Holen Sie es auf folgende Weise von der Fabrik ab:

```
yourCar = [Car new];    neues Auto abholen
```

Sie senden der Klasse `Car` (dem Adressaten) eine Nachricht, in der Sie sie dazu auffordern, Ihnen ein neues Auto zu geben. Das resultierende Objekt (das Ihr einzigartiges Auto darstellt) wird dann in der Variable `yourCar` gespeichert. Von nun an kann `yourCar` als Referenz Ihrer Instanz des Autos verwendet werden, das Sie von der Fabrik abgeholt haben.

Da Sie zur Fabrik gingen, um das Auto abzuholen, wird die Methode **NEW** *Fabrik-*, *Factory-* oder *Klassenmethode* genannt. Die weiteren Aktionen an Ihrem neuen Auto sind Instanzmethoden, weil sie sich auf Ihr Auto beziehen. Die folgenden Beispiele zeigen Nachrichten, die Sie für Ihr Auto schreiben können:

```
[yourCar prep];     Ihr Auto für die erste Nutzung vorbereiten
[yourCar drive];    Ihr Auto fahren
[yourCar wash];     Ihr Auto waschen
[yourCar getGas];   Ihr Auto bei Bedarf betanken
[yourCar service];  Ihr Auto warten lassen

[yourCar topDown];  Wenn es sich um ein Cabrio handelt
[yourCar topUp];
currentMileage = [suesCar currentOdometer];
```

Das letzte Beispiel zeigt eine Instanzmethode, die Informationen zurückgibt – voraussichtlich den aktuellen Kilometerstand, wie ihn der Kilometerzähler anzeigt. Hier speichern wir diese Information in einer Variable in unserem Programm, die wir `currentMileage` nennen.

Ihre Schwester, Sue, kann die gleichen Methoden für ihre eigene Instanz eines Autos verwenden:

```
[suesCar drive];
[suesCar wash];
[suesCar getGas];
```

Die gleichen Methoden auf verschiedene Objekte anzuwenden, ist eines der grundlegenden Konzepte der objektorientierten Programmierung. Später erfahren Sie mehr dazu.

Sie werden in Ihren Programmen vermutlich nicht mit Autos arbeiten müssen. Ihre Objekte sind wahrscheinlich Dinge mit Bezug zu Computern, wie Fenster, Rechtecke, Textteile, vielleicht eine Berechnung oder eine Wiedergabeliste von Musiktiteln. Wie die für Ihre Autos verwendeten Methoden sehen sich Ihre Methoden vielleicht ähnlich, wie in den folgenden Beispielen:

```
[myWindow erase];              Das Fenster leeren
[myRect getArea];              Die Fläche des Rechtecks berechnen
[userText spellCheck];         Rechtschreibprüfung für einen Text durchführen
[deskCalculator clearEntry];   Den letzten Eintrag löschen
[favoritePlaylist showSongs]; Die Titel in einer Wiedergabeliste von
                               Lieblingsstücken anzeigen
[phoneNumber dial];            Eine Telefonnummer wählen
```

3.3 Eine Objective-C-Klasse zum Rechnen mit Brüchen

Nun ist es an der Zeit, tatsächlich eine Klasse in Objective-C zu definieren und zu lernen, wie Sie mit Instanzen der Klasse arbeiten können.

Wiederum lernen Sie zunächst den Ablauf. Die Folge davon ist, dass das eigentliche Programm nicht besonders sinnvoll erscheinen dürfte. Wir kommen später zu praxisnäheren Anwendungen.

Angenommen, Sie müssen ein Programm schreiben, das mit Brüchen arbeiten kann. Vielleicht müssen Sie sie addieren, subtrahieren, multiplizieren usw. Wenn Sie nichts über Klassen wüssten, könnten Sie mit einem einfachen Programm wie dem folgenden beginnen:

Programm 3.1

```
// Einfaches Programm zum Rechnen mit Brüchen
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int numerator = 1;
   int denominator = 3;
   NSLog (@"The fraction is %i/%i", numerator, denominator);

   [pool drain];
   return 0;
}
```

Programm 3.1 – Ausgabe

```
The fraction is 1/3
```

In Programm 3.1 wird der Bruch mithilfe seines Zählers (englisch *numerator*) und Nenners (*denominator*) dargestellt. Wenn der Autorelease-Pool erstellt worden ist, deklarieren die beiden Zeilen in `main` die Variablen `numerator` und `denominator` als Ganzzahlen und weisen ihnen Anfangswerte von `1` bzw. `3` zu. Dies entspricht dem folgenden Code:

```
int numerator, denominator;

numerator = 1;
denominator = 3;
```

Wir stellten den Bruch 1/3 dar, indem wir den Wert 1 in der Variable für den Zähler und 3 in der für den Nenner speichern. Wenn Sie eine Vielzahl von Brüchen in Ihrem Programm ablegen müssen, kann dies sehr umständlich sein. Jedes Mal, wenn Sie mit dem Bruch arbeiten wollen, müssen Sie sich auf die entsprechenden Zähler und Nenner beziehen, und mathematische Operationen an diesen Brüchen sind ebenso schwierig.

Es ist besser, wenn Sie einen Bruch als eine Dateneinheit definieren und sich mit nur einem Namen wie zum Beispiel `myFraction` gleichzeitig auf Zähler und Nenner beziehen. In Objective-C können Sie das tun, wobei zunächst eine neue Klasse zu definieren ist.

Programm 3.2 erfüllt die Funktion von Programm 3.1 mithilfe der neuen Klasse `Fraction`. Im Folgenden zeigen wir zunächst das Programm und danach eine ausführliche Erklärung, wie es funktioniert.

Programm 3.2

```
// Programm zum Rechnen mit Brüchen - Version mit Klassen

#import <Foundation/Foundation.h>

//---- Abschnitt @interface ----
@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}
-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
@end

//---- Abschnitt @implementation ----
@implementation Fraction
-(void) print
{
```

```
   NSLog (@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) n
{
   numerator = n;
}
-(void) setDenominator: (int) d
{
denominator = d;
}
@end

//---- Abschnitt program----
int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Fraction *myFraction;

   // Eine Instanz eines Bruchs erstellen
   myFraction = [Fraction alloc];
   myFraction = [myFraction init];

   // Bruch auf 1/3 setzen
   [myFraction setNumerator: 1];
   [myFraction setDenominator: 3];

   // Den Bruch mithilfe der Methode print anzeigen
   NSLog (@"The value of myFraction is:");
   [myFraction print];
   [myFraction release];

   [pool drain];
   return 0;
}
```

Programm 3.2 – Ausgabe

```
The value of myFraction is:
1/3
```

Wie Sie anhand der Kommentare in Programm 3.2 erkennen, ist das Programm in drei logisch voneinander abgesetzte Abschnitte unterteilt:

- **Abschnitt** `@interface`
- **Abschnitt** `@implementation`
- **Abschnitt** `program`

Der Abschnitt `@interface` beschreibt die Klasse, ihre Datenbestandteile und ihre Methoden, während der Abschnitt `@implementation` den eigentlichen Code enthält, der diese Methoden implementiert. Der Abschnitt `program` schließlich schließt den Programmcode ein, der dafür sorgt, dass das Programm seinen Zweck erfüllt.

Diese Abschnitte sind stets Teil eines jeden Programms in Objective-C, auch wenn Sie nicht immer jeden selbst schreiben müssen. Wie Sie sehen werden, wird jeder Abschnitt normalerweise in seiner eigenen Datei gespeichert. Vorerst wollen wir aber alle zusammen in einer einzigen Datei belassen.

3.4 Der Abschnitt @interface

Um eine neue Klasse zu definieren, müssen Sie mehrere Dinge tun. Erstens müssen Sie dem Objective-C-Compiler mitteilen, woher die Klasse stammt, d.h. Sie müssen ihre *Elternklasse* benennen. Zweitens müssen Sie festlegen, welche Art von Daten in den Objekten dieser Klasse gespeichert werden soll. Sie müssen also die Daten beschreiben, die die Mitglieder (Member) der Klasse enthalten werden. Diese Member werden *Instanzvariablen* genannt. Schließlich müssen Sie die Art der Handlungen definieren, also die *Methoden*, die beim Umgang mit Objekten dieser Klasse verwendet werden können. Dies alles geschieht in einem bestimmten Abschnitt des Programms, nämlich dem Abschnitt `@interface`. Im Folgenden sehen Sie sein allgemeines Format:

```
@interface NameDerNeuenKlasse: NameDerElternklasse
{
memberDeklarationen;
}
methodenDeklarationen;
@end
```

Es ist üblich, Klassennamen mit einem Großbuchstaben zu beginnen, auch wenn dies nicht erforderlich ist. Dies ermöglicht den Lesern Ihres Programms, Klassennamen von anderen Arten von Variablen zu unterscheiden, indem sie sich einfach den ersten Buchstaben des Namens ansehen. An dieser Stelle möchten wir kurz abschweifen, um die Bildung von Namen in Objective-C zu erläutern.

3.4.1 Namen auswählen

In Kapitel 2, *Programmierung in Objective-C*, haben Sie mehrere Variablen verwendet, um Integerwerte zu speichern. So haben Sie zum Beispiel die Variable `sum` in Programm 2.4 verwendet, um das Ergebnis der Addition der beiden Ganzzahlen 50 und 25 zu speichern.

Objective-C ermöglicht es Ihnen, in Variablen auch andere Datentypen als Integer zu speichern, solange die richtige Deklaration für die Variable erfolgt, bevor sie im Programm verwendet wird. Variablen können dazu verwendet werden, Fließkommazahlen, Zeichen und sogar Objekte zu speichern (oder genauer gesagt Referenzen auf Objekte).

Die Benennungsregeln sind recht einfach: Namen müssen mit einem Buchstaben oder einem Unterstrich (_) anfangen und mit jeder beliebigen Kombination von Groß- oder Kleinbuchstaben, Unterstrichen oder Ziffern 0–9 fortgesetzt werden. Die folgende Liste zeigt zulässige Namen:

```
> sum
> pieceFlag
> i
> myLocation
> numberOfMoves
> _sysFlag
> ChessBoard
```

Andererseits sind die folgenden Namen aus den genannten Gründen nicht zulässig:

- `sum$value` – **$ ist kein zulässiges Zeichen.**
- `piece flag` – **eingeschlossene Leerzeichen sind nicht zulässig.**
- `3Spencer` – **Namen können nicht mit einer Zahl beginnen.**
- `int` – **Dieses Wort ist reserviert.**

`int` kann nicht als Variablenname verwendet werden, weil dieses Wort eine besondere Bedeutung für den Objective-C-Compiler hat. Es wird als *reservierter Name* oder reserviertes Wort bezeichnet. Allgemein gilt, dass kein Name, der für den Objective-C-Compiler von besonderer Bedeutung ist, als Variablenname verwendet werden kann. In Anhang B, *Übersicht über Objective-C 2.0*, finden Sie eine vollständige Auflistung solcher reservierter Namen.

Denken Sie immer daran, dass Groß- und Kleinbuchstaben in Objective-C unterschieden werden. Aus diesem Grund stehen die Variablennamen `sum`, `Sum` und `SUM` jeweils für eine andere Variable. Beginnen Sie Klassennamen, wie bereits erwähnt, mit einem Großbuchstaben. Instanzvariablen, Objekte und Methodennamen fangen hingegen normalerweise mit Kleinbuchstaben an. Zur besseren Lesbarkeit werden Großbuchstaben wie in den folgenden Beispielen innerhalb von Namen verwendet, um den Beginn eines neuen Worts zu kennzeichnen:

- `AddressBook` – **Dies könnte ein Klassenname sein.**
- `currentEntry` – **Dies könnte ein Objekt bezeichnen.**
- `current_entry` – **Einige Programmierer verwenden Unterstriche zur Trennung von Wörtern.**
- `addNewEntry` – **Dies könnte der Name einer Methode sein.**

Wenn Sie sich für einen Namen entscheiden, nehmen Sie sich einen Rat zu Herzen: Seien Sie nicht nachlässig. Wählen Sie Namen, die die angestrebte Verwendung der Variable oder des Objekts widerspiegeln. Die Gründe dafür sind offensichtlich. Genau wie bei den Kommentaranweisungen können aussagekräftige Namen die Lesbarkeit eines Programms erheblich steigern, was sich bei der Fehlerbehebung und Dokumentation auszahlt. In der Tat kann dies die Dokumentation stark erleichtern, weil das Programm dann weitgehend selbsterklärend ist.

Hier zeigen wir nochmals den Abschnitt `@interface` aus Programm 3.2:

```
//---- Abschnitt @interface ----

@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

Der Name der neuen Klasse lautet `Fraction`, und ihre Elternklasse ist `NSObject`. (In Kapitel 8, *Vererbung*, erläutern wir Elternklassen genauer.) Die Klasse `NSObject` wird in der Datei `NSObject.h` definiert, die immer automatisch in Ihrem Programm enthalten ist, wenn Sie `Foundation.h` importieren.

3.4.2 Instanzvariablen

Der Abschnitt `memberDeklarationen` legt fest, welche Arten von Daten in einem Bruch der Klasse `Fraction` gespeichert werden und wie diese Datentypen heißen. Wie Sie sehen, ist dieser Abschnitt für sich allein von geschweiften Klammern umschlossen. Für Ihre Klasse `Fraction` besagen diese Deklarationen, dass ein Objekt der Klasse aus zwei ganzzahligen Membern besteht, wobei `numerator` für den Zähler und `denominator` für den Nenner steht:

```
int numerator;
int denominator;
```

Die in diesem Abschnitt deklarierten Member werden Instanzvariablen genannt. Wie Sie sehen werden, ruft die Erstellung eines neuen Objekts immer auch einen neuen und eindeutigen Satz von Instanzvariablen hervor. Wenn Sie zwei Brüche der Klasse `Fraction` haben, `fracA` und `fracB`, weist jeder seinen eigenen Satz Instanzvariablen auf. Das heißt, `fracA` und `fracB` haben jeweils einen eigenen `numerator` und `denominator`. Objective-C verfolgt dies automatisch für Sie, was zu den angenehmen Dingen bei der Arbeit mit Objekten zählt.

3.4.3 Klassen- und Instanzmethoden

Sie müssen Methoden definieren, um mit Ihren Klassen arbeiten zu können. Beispielsweise müssen Sie in der Lage sein, den Wert eines Bruchs auf einen bestimmten Wert setzen zu können. Da Sie keinen direkten Zugriff auf die interne Darstellung eines Bruchs haben (oder mit anderen Worten auf seine Instanzvariablen), müssen Sie Methoden schreiben, um den Zähler und den Nenner festzulegen. Sie brauchen auch eine Methode namens `print`, die den Wert eines Bruchs wiedergibt. Im Folgenden sehen Sie, wie die Deklaration für die Methode `print` in der Schnittstellendatei aussieht:

```
-(void) print;
```

Das vorangestellte Minuszeichen (-) teilt dem Objective-C-Compiler mit, dass die Methode eine Instanzmethode ist. Die einzige Alternative ist ein Pluszeichen (+), das eine Klassenmethode anzeigt. Definitionsgemäß führt eine Klassenmethode die Tätigkeiten an der Klasse selbst aus, indem sie zum Beispiel eine neue Instanz der Klasse erstellt. Dies entspricht der Produktion eines neuen Autos, wobei das Auto die Klasse darstellt. Wenn Sie ein neues herstellen wollen, ist dies eine Klassenmethode.

Eine Instanzmethode führt eine Handlung an einer bestimmten Instanz einer Klasse aus, wie ihren Wert festzulegen, ihren Wert abzurufen, ihren Wert anzuzeigen usw. Im Beispiel der Autos können Sie ein Auto betanken, nachdem Sie es hergestellt haben. Die Tätigkeit des Betankens wird an einem bestimmten Auto ausgeführt, daher entspricht dies einer Instanzmethode.

Rückgabewerte

Wenn Sie eine neue Methode deklarieren, müssen Sie dem Objective-C-Compiler mitteilen, ob die Methode einen Wert zurückgibt, und falls ja, welche Art von Wert dies ist. Sie tun dies, indem Sie den Rückgabetyp in Klammern nach dem einführenden Minus- oder Pluszeichen angeben. Somit legt diese Deklaration fest, dass die Instanzmethode mit dem Namen `retrieveNumerator` einen Integerwert zurückgibt:

```
-(int) retrieveNumerator;
```

Entsprechend deklariert die folgende Zeile eine Methode, die einen Wert vom Typ `double` zurückgibt. (Mehr über diesen Datentyp erfahren Sie in Kapitel 4, *Datentypen und Ausdrücke*.)

```
-(double) retrieveDoubleValue;
```

Ein Wert wird von einer Methode in Objective-C mithilfe der Anweisung `return` in gleicher Weise zurückgegeben, wie wir einen Wert von `main` in den voranstehenden Programmbeispielen zurückgegeben haben.

Wenn die Methode keinen Wert zurückgibt, zeigen Sie dies wie im folgenden Beispiel durch den Typ `void` an:

```
-(void) print;
```

Damit deklarieren Sie eine Instanzmethode namens `print`, die keinen Wert zurückgibt. In einem solchen Fall müssen Sie keine `return`-Anweisung am Ende Ihrer Methode ausführen. Alternativ können Sie `return` wie im folgenden Beispiel ohne festgelegten Wert ausführen:

```
return;
```

Sie müssen für Ihre Methoden keinen Rückgabetyp festlegen, wobei dies aber als gute Programmierpraxis gilt. Wenn Sie keinen Typ festlegen, wird standardmäßig `id` vorgegeben. Mehr zu diesem Datentyp erfahren Sie in Kapitel 9, *Polymorphismus, dynamische Typisierung und dynamische Bindung*. Grundsätzlich können Sie den Typ `id` für Bezüge auf jeden beliebigen Objekttyp verwenden.

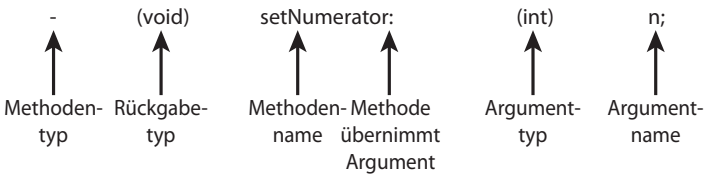
Methodenparameter

Im Abschnitt `@interface` aus Programm 3.2 werden noch zwei andere Methoden deklariert:

```
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
```

Beides sind Instanzmethoden, die keinen Wert zurückgeben. Jede Methode nimmt ein Integerargument an, was durch `(int)` vor seinem Namen angezeigt wird. Im Fall von `setNumerator` ist der Name des Arguments `n`. Diese Bezeichnung ist willkürlich gewählt und stellt den Namen dar, den die Methode verwendet, um auf das Argument zu verweisen. Aus diesem Grund legt die Deklaration von `setNumerator` fest, dass ein Integerargument namens `n` an die Methode übergeben und kein Wert zurückgegeben wird. Dies gilt ähnlich für `setDenominator`, außer dass der Name des Arguments hier `d` lautet.

Beachten Sie die Syntax der Deklaration für diese Methoden. Jeder Methodenname endet mit einem Doppelpunkt, wodurch der Objective-C-Compiler weiß, dass die Methode ein Argument erwartet. Dann wird der Typ des Arguments innerhalb von Klammern festgelegt, mehr oder weniger in der gleichen Art wie der Rückgabetyp für die Methode selbst. Schließlich wird der symbolische Name angegeben, der zur Identifizierung dieses Arguments in der Methode verwendet wird. Die gesamte Deklaration wird mit einem Semikolon beendet. Abbildung 3.1 veranschaulicht diese Syntax.



▶ *Abbildung 3.1: Eine Methode deklarieren*

Wenn eine Methode ein Argument benötigt, fügen Sie auch dann einen Doppelpunkt an ihren Namen an, wenn Sie auf sie verweisen. Aus diesem Grund ist `setNumerator:` und `setDenominator:` die richtige Art, diese beiden Methoden zu bezeichnen, wobei jede von ihnen ein einziges Argument annimmt. Dass der Methode `print` kein Doppelpunkt angefügt wird, zeigt an, dass diese Methode keine Argumente annimmt. In Kapitel 7, *Weiteres zu Klassen*, werden Sie sehen, wie Methoden angeführt werden, die mehr als ein Argument annehmen.

3.5 Der Abschnitt @implementation

Wie bereits erwähnt, enthält der Abschnitt `@implementation` den tatsächlichen Code für die Methoden, die Sie im Abschnitt `@interface` deklariert haben. Um die Terminologie zu verdeutlichen: Man sagt, dass Methoden im Abschnitt `@interface` *deklariert* und im Abschnitt `@implementation` *definiert* (d.h. im eigentlichen Sinne kodiert) werden.

Das allgemeine Format für den Abschnitt `@implementation` zeigt das folgende Beispiel:

```
@implementation NameDerNeuenKlasse
methodenDefinitionen;
@end
```

`NameDerNeuenKlasse` ist der gleiche Name, der für die Klasse im Abschnitt `@interface` verwendet wurde. Sie können danach einen Doppelpunkt verwenden, dem Sie den Namen der Elternklasse folgen lassen, wie wir es im Abschnitt `@interface` getan haben:

```
@implementation Fraction: NSObject
```

Dies ist aber optional und wird üblicherweise nicht getan.

Der Teil *methodenDefinitionen* des Abschnitts `@implementation` enthält den Programmcode für jede Methode, die im Abschnitt `@interface` festgelegt wurde. Ähnlich wie dort beginnt die Definition jeder Methode damit, ihren Typ (Klassen- oder Instanzmethode), ihren Rückgabetyp, ihre Argumente und deren Typen anzugeben. Allerdings folgt anstelle eines Semikolons am Zeilenende der Code für die Methode, eingeschlossen in geschweifte Klammern.

Betrachten Sie den Abschnitt `@implementation` aus Programm 3.2:

```
//---- Abschnitt @implementation ----
@implementation Fraction
-(void) print
-(void) setDenominator: (int) d
{
   denominator = d;
}
@end
```

Die Methode `print` verwendet `NSLog`, um die Werte der Instanzvariablen `numerator` und `denominator` anzuzeigen. Aber auf welchen Zähler und Nenner bezieht sich diese Methode? Sie verweist auf die Instanzvariablen, die in dem Objekt enthalten sind, und welche die Adressaten der Nachricht sind. Dies ist ein wichtiges Konzept, auf das wir in Kürze zurückkommen.

Die Methode `setNumerator:` speichert das Integerargument, das Sie `n` genannt haben, in der Instanzvariable `numerator`. Genauso speichert `setDenominator:` den Wert des Arguments `d` in der Instanzvariable `denominator`.

3.6 Der Abschnitt program

Der Abschnitt `program` enthält den Code, der Ihr jeweiliges Problem löst. Er kann sich bei Bedarf über viele Dateien erstrecken. Irgendwo muss es eine Routine mit dem Namen `main` geben, wie wir bereits angemerkt haben. Dort beginnt Ihr Programm stets mit der Ausführung. Hier zeigen wir erneut den Abschnitt `program` aus Programm 3.2:

```
//---- Abschnitt program ----

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Fraction *myFraction;
   // Eine Instanz eines Bruchs erstellen
   myFraction = [Fraction alloc];
   myFraction = [myFraction init];
   // Bruch auf 1/3 setzen
   [myFraction setNumerator: 1];
   [myFraction setDenominator: 3];
   // Den Bruch mithilfe der Methode print anzeigen
   NSLog (@"The value of myFraction is:");
   [myFraction print];

   [myFraction release];
   [pool drain];

   return 0;
}
```

Innerhalb von `main` definieren Sie mit der folgenden Codezeile eine Variable namens `myFraction`:

```
Fraction *myFraction;
```

Diese Zeile besagt, dass `myFraction` ein Objekt vom Typ `Fraction` ist, was bedeutet, dass `myFraction` verwendet wird, um Werte aus Ihrer neuen Klasse `Fraction` zu speichern. Das Sternchen (*) vor `myFraction` ist erforderlich, aber machen Sie sich keine Gedanken über seinen Zweck. Technisch besagt es, dass `myFraction` eigentlich eine Referenz (oder ein Zeiger) auf einen Bruch der Klasse `Fraction` ist.

Jetzt, da Sie ein Objekt haben, um einen Bruch zu speichern, müssen Sie einen erstellen, genauso als wenn Sie die Fabrik bitten, Ihnen ein neues Auto zu bauen. Dies geschieht über die folgende Zeile:

```
myFraction = [Fraction alloc];
```

`alloc` steht für »allocate«, also zuweisen. Sie wollen einem neuen Bruch Speicherplatz zuweisen. Der folgende Ausdruck sendet eine Nachricht an Ihre soeben erstellte Klasse `Fraction`:

```
[Fraction alloc]
```

Sie bitten die Klasse `Fraction`, die Methode `alloc` anzuwenden, aber Sie haben diese Methode nie definiert. Wo also kommt sie her? Die Methode wurde von einer Elternklasse vererbt. Kapitel 8, *Vererbung*, befasst sich ausführlich mit diesem Thema.

Wenn Sie `alloc` an eine Klasse senden, gibt diese eine neue Instanz zurück. In Programm 3.2 wird der zurückgegebene Wert in Ihrer Variable `myFraction` gespeichert. Die `alloc`-Methode sorgt zuverlässig dafür, dass alle Instanzvariablen eines Objekts auf null gesetzt werden. Dies bedeutet

allerdings nicht, dass das Objekt für die Verwendung ordnungsgemäß initialisiert wird. Sie müssen ein Objekt nach seiner Zuweisung über `alloc` auch noch initialisieren.

Dies geschieht in Programm 3.2 über die nächste Anweisung, die wie folgt aussieht:

```
myFraction = [myFraction init];
```

Wieder verwenden Sie hier eine Methode, die Sie nicht selbst geschrieben haben. Die Methode `init` initialisiert die Instanz einer Klasse. Beachten Sie, dass Sie die Nachricht `init` an `myFraction` senden. Sie wollen hier also ein bestimmtes Objekt der Klasse `Fraction` initialisieren, daher senden Sie die Nachricht nicht an die Klasse, sondern an eine Instanz. Achten Sie darauf, dass Sie diesen Punkt wirklich verstehen, bevor Sie fortfahren.

Die Methode `init` gibt ebenfalls einen Wert zurück – und zwar das initialisierte Objekt. Sie speichern den Rückgabewert in Ihrer Variable `myFraction`.

Die hier zweizeilig geschriebene Befehlsabfolge für die Zuweisung und Initialisierung einer neuen Klasseninstanz erfolgt in Objective-C so oft, dass die beiden Nachrichten im Normalfall wie folgt zusammengefasst werden:

```
myFraction = [[Fraction alloc] init];
```

Als Erstes betrachten wir den inneren Nachrichtenausdruck:

```
[Fraction alloc]
```

Wie Sie wissen, ist das Ergebnis dieses Nachrichtenausdrucks der eigentliche Bruch, der zugewiesen wird. Anstatt das Ergebnis der Zuweisung wie zuvor in einer Variable zu speichern, wenden Sie die Methode `init` direkt auf sie an. Daher weisen Sie wiederum zunächst einen neuen Bruch zu und initialisieren ihn dann. Das Ergebnis der Initialisierung wird dann der Variable `myFraction` zugewiesen.

Eine weitere häufig verwendete Kurzschreibweise besteht schließlich darin, Zuweisung und Initialisierung wie im folgenden Beispiel direkt in die Deklarationszeile einzubinden:

```
Fraction *myFraction = [[Fraction alloc] init];
```

Wir verwenden diesen Programmierstil häufig im weiteren Verlauf dieses Buchs, daher ist es wichtig, dass Sie ihn verstehen. Sie haben ihn schon in allen bisherigen Programmen bei der Zuweisung des Autorelease-Pools gesehen:

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

Hier wird die Nachricht `alloc` an die Klasse `NSAutoreleasePool` gesendet, wodurch eine neue Instanz angefordert wird. Dann wird die Nachricht `init` an das soeben erstellte Objekt gesendet, um es zu initialisieren.

Nun sind Sie in der Lage, den Wert Ihres Bruchs aus Programm 3.2 festzulegen. Die folgenden Programmzeilen sorgen dafür:

```
// Bruch auf 1/3 setzen

[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

Die erste Anweisung sendet die Nachricht `setNumerator:` an `myFraction`. Das übergebene Argument ist der Wert `1`. Die Steuerung wird dann an die Methode `setNumerator:` übergeben, die Sie für Ihre Klasse `Fraction` definiert haben. Objective-C weiß, dass die Methode aus dieser Klasse verwendet werden muss, da `myFraction` ein Objekt der Klasse `Fraction` ist.

Innerhalb der Methode `setNumerator:` wird der übergebene Wert von `1` in der Variable `n` gespeichert. Die einzige Programmzeile in dieser Methode speichert diesen Wert in der Instanzvariable `numerator`. Somit haben Sie den Zähler von `myFraction` erfolgreich auf 1 gesetzt.

Es folgt die Nachricht, die die Methode `setDenominator:` für `myFraction` aufruft. Das Argument `3` wird der Variable `d` innerhalb der Methode `setDenominator:` zugewiesen. Dieser Wert wird dann in der Instanzvariablen `denominator` gespeichert, wodurch die Zuweisung des Werts 1/3 zu `myFraction` abgeschlossen wird. Jetzt sind Sie in der Lage, den Wert Ihres Bruchs anzeigen zu lassen, was Sie mit den folgenden Zeilen aus Programm 3.2 tun:

```
// Den Bruch mithilfe der Methode print anzeigen
NSLog (@"The value of myFraction is:");
[myFraction print];
```

Der Aufruf von `NSLog` zeigt einfach den folgenden Text an:

```
The value of myFraction is:
```

Der folgende Nachrichtenausdruck ruft die Methode `print` auf:

```
[myFraction print];
```

In dieser Methode werden die Werte der Instanzvariablen `numerator` und `denominator` mit einem Schrägstrich voneinander getrennt angezeigt.

Die Nachricht im Programm leert oder löscht den Speicher, der für das Objekt von `Fraction` genutzt wurde:

```
[myFraction release];
```

Dies ist ein wesentlicher Bestandteil eines guten Programmierstils. Jedes Mal, wenn Sie ein neues Objekt erstellen, bitten Sie darum, ihm Speicher einzuräumen. Wenn Sie nicht mehr an diesem Objekt arbeiten, sind Sie für das Leeren des Speichers verantwortlich, der dafür verwendet wurde. Zwar wird der Speicher in der Tat in jedem Fall gelöscht, wenn Ihr Programm endet. Wenn Sie aber beginnen, komplexere Anwendungen zu entwickeln, arbeiten Sie am Ende mit Hunderten (oder Tausenden) von Objekten, die eine große Menge an Speicher beanspruchen. Darauf zu warten, dass das Programm endet, bevor der Speicher gelöscht wird, stellt Verschwendung von Speicherplatz dar, kann die Ausführung Ihres Programms verlangsamen und ist kein guter Programmierstil. Sie sollten sich daher angewöhnen, Speicher zu leeren, sobald dies möglich ist.

Das Laufzeitsystem von Apple enthält einen Mechanismus, der *Garbage Collection* genannt wird und für eine automatische Speicherbereinigung sorgt. Es ist allerdings am besten, Ihre Speichernutzung selbst zu steuern, anstatt sich auf diesen automatisierten Mechanismus zu verlassen. In der Tat können Sie sich auf die Garbage Collection nicht verlassen, wenn Sie für bestimmte Plattformen programmieren, auf denen sie nicht unterstützt wird, wie zum Beispiel das iPhone. Aus diesem Grund kommen wir erst wieder sehr viel weiter hinten in diesem Buch auf dieses Thema zurück.

Es scheint, als müssten Sie sehr viel mehr Code schreiben, wenn Sie in Programm 3.2 nachahmen wollen, was Sie mit Programm 3.1 bewerkstelligen konnten. Das trifft bei diesem einfachen Beispiel zu, doch das höchste Ziel bei der Arbeit mit Objekten liegt darin, das Schreiben, Pflegen und Erweitern Ihrer Programme zu erleichtern. Sie werden dies später erkennen.

Das letzte Beispiel in diesem Kapitel zeigt, wie Sie mit mehr als einem Bruch in Ihrem Programm arbeiten können. In Programm 3.3 setzen Sie einen Bruch auf 2/3, einen weiteren auf 3/7 und zeigen beide an.

Programm 3.3

```
// Programm zum Arbeiten mit Brüchen - Fortsetzung

#import <Foundation/Foundation.h>
//---- Abschnitt @interface ----
@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//---- Abschnitt @implementation ----

@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) n
{
   numerator = n;
}
-(void) setDenominator: (int) d
{
   denominator = d;
}
@end

//---- Abschnitt program ----
int main (int argc, char *argv[])
{
```

```
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *frac1 = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // Ersten Bruch auf 2/3 setzen

    [frac1 setNumerator: 2];
    [frac1 setDenominator: 3];

    // Zweiten Bruch auf 3/7 setzen

    [frac2 setNumerator: 3];
    [frac2 setDenominator: 7];

    // Brüche anzeigen

    NSLog (@"Frist fraction is:");
    [frac1 print];

    NSLog (@"Second fraction is:");
    [frac2 print];

    [frac1 release];
    [frac2 release];

    [pool drain];
    return 0;
}
```

Programm 3.3 – Ausgabe

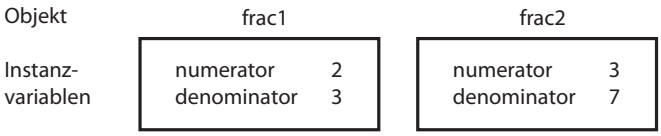
```
First fraction is:
2/3
Second fraction is:
3/7
```

Die Abschnitte `@interface` und `@implementation` werden aus Programm 3.2 unverändert übernommen. Das Programm erstellt zwei `Fraction`-Objekte, `frac1` und `frac2`, und weist dann dem ersten Bruch den Wert 2/3 zu und dem zweiten 3/7. Vergegenwärtigen Sie sich, dass die Instanzvariable `frac1` die Instanzvariable `numerator` auf 2 setzt, wenn die Methode `setNumerator:` auf `frac1` angewandt wird, um den Zähler dieses Bruchs auf 2 zu setzen. Wenn `frac2` die gleiche Methode verwendet, um den Zähler dieses Bruchs als 3 festzulegen, wird die Instanzvariable `numerator` von `frac2` auf diesen Wert gesetzt. Jedes Mal, wenn Sie ein neues Objekt erstellen, erhält dieses seinen eigenen, separaten Satz von Instanzvariablen. Abbildung 3.2 veranschaulicht dies.

Abhängig davon, welchem Objekt die Nachricht zugesandt wird, werden die richtigen Instanzvariablen referenziert. Daher wird im folgenden Code stets `numerator` von `frac1` referenziert, wenn `setNumerator:` den Namen `numerator` in der Methode verwendet:

```
[frac1 setNumerator: 2];
```

Dies liegt daran, dass `frac1` der Empfänger der Nachricht ist.



▶ *Abbildung 3.2: Eindeutige Instanzvariablen*

3.7 Zugriff auf Instanzvariablen und Datenkapselung

Sie haben gesehen, wie die Methoden für Brüche auf die beiden Instanzvariablen `numerator` und `denominator` direkt namentlich zugreifen können. Eine Instanzmethode kann immer direkt auf ihre Instanzvariablen zugreifen. Eine Klassenmethode kann das jedoch nicht, da sie nur mit der Klasse selbst umgeht, nicht mit den Instanzen der Klasse (denken Sie darüber einen Moment lang nach). Aber was, wenn Sie auf die Instanzvariablen von anderen Stellen aus zugreifen wollten – zum Beispiel aus der Routine `main` heraus? Sie können das nicht direkt tun, da sie verborgen sind. Diese Tatsache ist ein wesentliches Prinzip, das *Datenkapselung* heißt. Es ermöglicht jemandem, der Klassendefinitionen schreibt, diese zu erweitern und abzuändern, ohne sich Gedanken darüber machen zu müssen, ob andere Programmierer (also Benutzer der Klasse) an den internen Einzelheiten der Klasse herumpfuschen. Datenkapselung bildet eine komfortable Isolierungsschicht zwischen dem Programmierer und dem Entwickler der Klasse.

Sie können auf Ihre Instanzvariablen sauber zugreifen, indem Sie besondere Methoden schreiben, um ihre Werte abzurufen. Sie erstellen zum Beispiel zwei neue Methoden, die Sie – treffend genug – `numerator` und `denominator` nennen, um auf die entsprechenden Instanzvariablen der `Fraction`-Instanz zuzugreifen, die der Empfänger der Nachricht ist. Das Ergebnis ist der entsprechende Integerwert, den Sie anschließend zurückgeben. Hier zeigen wir die Deklarationen für Ihre beiden neuen Methoden:

```
-(int) numerator;
-(int) denominator;
```

Und hier die Definitionen:

```
-(int) numerator
{
   return numerator;
}
-(int) denominator
{
   return denominator;
}
```

Beachten Sie, dass die Namen der Methoden und der Instanzvariablen, auf die sie zugreifen, identisch sind. Dies ist nicht problematisch; in der Tat ist dies gang und gäbe. Programm 3.4 testet Ihre beiden neuen Methoden.

Programm 3.4

```
// Programm zum Zugriff auf Instanzvariablen - Fortsetzung

#import <Foundation/Foundation.h>
//---- @interface section ----

@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end

//---- Abschnitt @implementation ----

@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) n
{
   numerator = n;
}
-(void) setDenominator: (int) d
{
   denominator = d;
}
-(int) numerator
{
   return numerator;
}
-(int) denominator
{
```

```
   return denominator;
}
@end

//---- Abschnitt program ----

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Fraction *myFraction = [[Fraction alloc] init];

   // Bruch auf 1/3 setzen

   [myFraction setNumerator: 1];
   [myFraction setDenominator: 3];

   // Den Bruch mit den beiden neuen Methoden anzeigen

   NSLog (@"The value of myFraction is: %i/%i",
   [myFraction numerator], [myFraction denominator]);
   [myFraction release];
   [pool drain];

   return 0;
}
```

Programm 3.4 – Ausgabe

```
The value of myFraction is: 1/3
```

Die folgende `NSLog`-Anweisung zeigt das Ergebnis des Vorgangs, dass zwei Nachrichten an `myFraction` gesendet werden: der erste zum Abruf des Zählerwerts, der zweite für den Wert des Nenners:

```
NSLog (@"The value of myFraction is: %i/%i",
[myFraction numerator], [myFraction denominator]);
```

Methoden, die die Werte von Instanzvariablen festlegen, werden häufig als *Änderungs-* oder *Set-Methoden* bezeichnet, und Methoden, die die Werte von Instanzvariablen abrufen, als *Abfrage-* oder *Get-Methoden*. Für die Klasse `Fraction` sind `setNumerator:` und `setDenominator:` die Set-, `numerator` und `denominator` die Get-Methoden.

> **Hinweis**
>
> Bald werden Sie eine praktische Eigenschaft von Objective-C 2.0 kennen lernen, die automatische Erstellung von Set- und Get-Methoden.

Außerdem gibt es eine Methode namens `new`, die die Aktionen von `alloc` und `init` in sich vereint. So kann auch die folgende Zeile verwendet werden, um eine neue Klasse namens `Fraction` zuzuweisen und zu initialisieren:

```
Fraction *myFraction = [Fraction new];
```

Im Allgemeinen ist es besser, die Zuweisung und Initialisierung in zwei Schritten durchzuführen, um sich klar zu machen, dass zwei voneinander unterschiedliche Ereignisse stattfinden: Erst erstellen Sie ein neues Objekt und dann initialisieren Sie es.

3.8 Zusammenfassung

Nun wissen Sie, wie Sie Ihre eigene Klasse definieren, Objekte oder Instanzen dieser Klasse erstellen und Nachrichten an diese Objekte senden können. Wir kehren zur Klasse `Fraction` in späteren Kapiteln zurück. Dort werden Sie lernen, wie Sie Ihren Methoden mehrere Argumente zuweisen, wie Sie Ihre Klassendefinitionen in separate Dateien teilen und wie Sie grundlegende Konzepte wie Vererbung und dynamische Bindung verwenden. Jetzt ist es allerdings an der Zeit, mehr über Datentypen und das Schreiben von Ausdrücken in Objective-C zu erfahren. Machen Sie sich aber zunächst an die folgenden Übungen, um Ihr Verständnis der wichtigen Punkte zu prüfen, die in diesem Kapitel behandelt wurden.

3.9 Übungen

1. Welche der folgenden Namen sind ungültig? Warum?

   ```
   Int            playNextSong    6_05
   _calloc        Xx              alphaBetaRoutine
   clearScreen    _1312           z
   ReInitialize   _               A$
   ```

2. Wählen Sie entsprechend dem Beispiel mit den Autos in diesem Kapitel ein Objekt, das Sie jeden Tag benutzen. Geben Sie für dieses Objekt eine Klasse an und schreiben Sie fünf Tätigkeiten auf, die Sie mit diesem Objekt verrichten.

3. Schreiben Sie die in Übung 2 aufgestellte Liste in ein neues Format um und verwenden Sie dabei die folgende Syntax:

   ```
   [instance method];
   ```

4. Stellen Sie sich vor, dass Sie zusätzlich zu Ihrem Auto auch ein Boot und ein Motorrad besitzen. Führen Sie die Tätigkeiten auf, diese Sie mit jedem der genannten Gegenstände ausführen. Gibt es dabei Überschneidungen?

5. Stellen Sie sich auf der Grundlage von Übung 4 vor, dass Sie eine Klasse mit dem Namen `Vehicle` und ein Objekt namens `myVehicle` haben, das entweder `Car`, `Motorcycle` oder `Boat` sein kann. Sie können dazu Folgendes geschrieben haben:

```
[myVehicle prep];
[myVehicle getGas];
[myVehicle service];
```

Sehen Sie Vorteile darin, eine Tätigkeit auf ein Objekt anwenden zu können, das einer von mehreren möglichen Klassen entstammt?

6. In einer prozeduralen Sprache wie C denken Sie an Tätigkeiten und schreiben dann Code, mit dessen Hilfe diese Aktionen an verschiedenen Objekten ausgeführt werden können. In unserem Autobeispiel könnten Sie in C eine Prozedur für das Waschen eines Beförderungsmittels schreiben und dann innerhalb dieser Prozedur Code dafür verfassen, wie beim Waschen eines Autos, eines Boots, eines Motorrads usw. vorzugehen ist. Nun nehmen wir an, dass Sie diesen Ansatz wählen und ein neues Transportmittel hinzufügen wollen (siehe dazu die vorangegangene Übung). Sehen Sie Vor- oder Nachteile in der Verwendung dieses prozeduralen Ansatzes gegenüber einem objektorientierten?
7. Definieren Sie eine Klasse mit dem Namen `XYPoint`, die eine kartesische Koordinate (x, y) enthält, wobei x und y ganzzahlig sind. Definieren Sie Methoden, um die x- und y-Koordinaten eines Punkts einzeln festlegen zu können, und rufen Sie ihre Werte ab. Schreiben Sie ein Programm in Objective-C, um Ihre neue Klasse zu implementieren, und testen Sie es.