Stephen G. Kochan

Third Edition

Programming in Objective-C 2.0

A complete introduction to the Objective-C language for Mac OS X and iPhone development

Developer's Library

Programming in Objective-C

Third Edition

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development
Luke Welling & Laura Thomson
ISBN 978-0-672-32916-6

MySQL
Paul DuBois
ISBN-13: 978-0-672-32938-8

Linux Kernel Development
Robert Love
ISBN-13: 978-0-672-32946-3

Python Essential Reference
David Beazley
ISBN-13: 978-0-672-32978-4

PostgreSQL
Korry Douglas
ISBN-13: 978-0-672-32756-2

C++ Primer Plus
Stephen Prata
ISBN-13: 978-0321-77640-2

Developer's Library books are available in print and in electronic formats at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

Developer's Library
informit.com/devlibrary

Programming in Objective-C

Third Edition

Stephen G. Kochan

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Programming in Objective-C, Third Edition

ISBN-13: 978-0-321-71139-7
ISBN-10: 0-321-71139-4

Library of Congress Cataloging-in-Publication Data:
Kochan, Stephen G.
Programming in objective-C / Stephen G. Kochan. – 3rd ed.
p. cm.
Includes index.
ISBN 978-0-321-71139-7 (pbk.)
1. Objective-C (Computer program language) 2. Object-oriented programming (Computer science) 3. Macintosh (Computer)–Programming.
I. Title.
QA76.64.K655 2011
005.1'17–dc23

2011015714

Printed in the United States of America

First Printing, June 2011

Trademarks

Warning and Disclaimer

Bulk Sales

Acquisitions Editor
Mark Taber

Development Editor
Michael Thurston

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Indexer
Larry Sweazy

Proofreader
Kathy Ruiz

Technical Editor
Michael Trent

Publishing Coordinator
Vanessa Evans

Designer
Gary Adair

Compositor
Mark Shirar

❖

To Roy and Ve, two people whom I dearly miss

❖

Contents at a Glance

Table of Contents

About the Author

Stephen Kochan is the author and coauthor of several bestselling titles on the C language, including *Programming in C* (Sams, 2004), *Programming in ANSI C* (Sams, 1994), and *Topics in C Programming* (Wiley, 1991), and several Unix titles, including *Exploring the Unix System* (Sams, 1992) and *Unix Shell Programming* (Sams, 2003). He has been programming on Macintosh computers since the introduction of the first Mac in 1984, and he wrote *Programming C for the Mac* as part of the Apple Press Library. In 2003 Kochan wrote *Programming in Objective-C* (Sams, 2003), and followed that with another Mac-related title, *Beginning AppleScript* (Wiley, 2004).

About the Technical Reviewer

Wendy Mui is a programmer and software development manager in the San Francisco Bay Area. After learning Objective-C from the second edition of Steve Kochan's book, she landed a job at Bump Technologies, where she put her programming skills to good use working on the client app and the API/SDK for Bump's third party developers. Prior to her iOS experience, Wendy spent her formative years at Sun and various other tech companies in Silicon Valley and San Francisco. She got hooked on programming while earning a B.A. in Mathematics from University of California Berkeley. When not working, Wendy is pursuing her 4th Dan Tae Kwon Do black belt.

Michael Trent has been programming in Objective-C since 1997—and programming Macs since well before that. He is a regular contributor to Steven Frank's www.cocoadev.com Web site, a technical reviewer for numerous books and magazine articles, and an occasional dabbler in Mac OS X open source projects. Currently, he is using Objective-C and Apple Computer's Cocoa frameworks to build professional video applications for Mac OS X. Michael holds a Bachelor of Science degree in computer science and a Bachelor of Arts degree in music from Beloit College of Beloit, Wisconsin. He lives in Santa Clara, California, with his lovely wife, Angela.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write Mark directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address.

E-mail: feedback@developers-library.info
Mail: Reader Feedback
Addison-Wesley Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

3

Classes, Objects, and Methods

In this chapter, you'll learn about some key concepts in object-oriented programming and start working with classes in Objective-C. You'll need to learn a little bit of terminology, but we keep it fairly informal. We also cover only some of the basic terms here because you can easily get overwhelmed. Refer to Appendix A, "Glossary," at the end of this book, for more precise definitions of these terms.

What Is an Object, Anyway?

An object is a thing. Think about object-oriented programming as a thing and something you want to do to that thing. This is in contrast to a programming language such as C, known as a procedural programming language. In C, you typically think about what you want to do first and then you worry about the objects, almost the opposite of object orientation.

Consider an example from everyday life. Let's assume that you own a car, which is obviously an object, and one that you own. You don't have just any car; you have a particular car that was manufactured in a factory, maybe in Detroit, maybe in Japan, or maybe someplace else. Your car has a vehicle identification number (VIN) that uniquely identifies that car here in the United States.

In object-oriented parlance, your particular car is an *instance* of a car. Continuing with the terminology, `car` is the name of the *class* from which this instance was created. So each time a new car is manufactured, a new instance from the class of cars is created, and each instance of the car is referred to as an *object*.

Your car might be silver, have a black interior, be a convertible or hardtop, and so on. Additionally, you perform certain actions with your car. For example, you drive your car, fill it with gas, (hopefully) wash it, take it in for service, and so on. Table 3.1 depicts this.

Table 3.1 **Actions on Objects**

Object	What You Do with It
[Your car]	Drive it
	Fill it with gas
	Wash it
	Service it

The actions listed in Table 3.1 can be done with your car, and they can be done with other cars as well. For example, your sister drives her car, washes it, fills it with gas, and so on.

Instances and Methods

A unique occurrence of a class is an instance, and the actions that are performed on the instance are called *methods*. In some cases, a method can be applied to an instance of the class or to the class itself. For example, washing your car applies to an instance (in fact, all the methods listed in Table 3.1 can be considered instance methods). Finding out how many types of cars a manufacturer makes would apply to the class, so it would be a class method.

Suppose you have two cars that came off the assembly line and are seemingly identical: They both have the same interior, same paint color, and so on. They might start out the same, but as each car is used by its respective owner, its unique characteristics or *properties* change. For example, one car might end up with a scratch on it and the other might have more miles on it. Each instance or object contains not only information about its initial characteristics acquired from the factory, but also its current characteristics. Those characteristics can change dynamically. As you drive your car, the gas tank becomes depleted, the car gets dirtier, and the tires get a little more worn.

Applying a method to an object can affect the *state* of that object. If your method is to "fill up my car with gas," after that method is performed, your car's gas tank will be full. The method then will have affected the state of the car's gas tank.

The key concepts here are that objects are unique representations from a class, and each object contains some information (data) that is typically private to that object. The methods provide the means of accessing and changing that data.

The Objective-C programming language has the following particular syntax for applying methods to classes and instances:

```
[ ClassOrInstance method ];
```

In this syntax, a left bracket is followed by the name of a class or instance of that class, which is followed by one or more spaces, which is followed by the method you want to perform. Finally, it is closed off with a right bracket and a terminating semicolon. When you ask a class or an instance to perform some action, you say that you are sending it a

message; the recipient of that message is called the *receiver*. So another way to look at the general format described previously is as follows:

```
[ receiver message ] ;
```

Let's go back to the previous list and write everything in this new syntax. Before you do that, though, you need to get your new car. Go to the factory for that, like so:

```
yourCar = [Car new];     get a new car
```

You send a `new` message to the `Car` class (the receiver of the message) asking it to give you a new car. The resulting object (which represents your unique car) is then stored in the variable `yourCar`. From now on, `yourCar` can be used to refer to your instance of the car, which you got from the factory.

Because you went to the factory to get the car, the method `new` is called a *factory* or *class* method. The rest of the actions on your new car will be instance methods because they apply to your car. Here are some sample message expressions you might write for your car:

```
[yourCar prep];          get it ready for first-time use
[yourCar drive];         drive your car
[yourCar wash];          wash your car
[yourCar getGas];        put gas in your car if you need it
[yourCar service];       service your car

[yourCar topDown];       if it's a convertible
[yourCar topUp];
currentMileage = [yourCar odometer];
```

This last example shows an instance method that returns information—presumably, the current mileage, as indicated on the odometer. Here we store that information inside a variable in our program called `currentMileage`.

Here's an example of where a method takes an *argument* that specifies a particular value that may differ from one method call to the next:

```
[yourCar setSpeed: 55];  set the speed to 55 mph
```

Your sister, Sue, can use the same methods for her own instance of a car:

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

Applying the same methods to different objects is one of the key concepts of object-oriented programming, and you'll learn more about it later.

You probably won't need to work with cars in your programs. Your objects will likely be computer-oriented things, such as windows, rectangles, pieces of text, or maybe even a calculator or a playlist of songs. And just like the methods used for your cars, your methods might look similar, as in the following:

`[myWindow erase];`	Clear the window
`theArea = [myRect area];`	Calculate the area of the rectangle
`[userText spellCheck];`	Spell-check some text
`[deskCalculator clearEntry];`	Clear the last entry
`[favoritePlaylist showSongs];`	Show the songs in a playlist of favorites
`[phoneNumber dial];`	Dial a phone number
`[myTable reloadData];`	Show the updated table's data
`n = [aTouch tapCount];`	Store the number of times the display was tapped

An Objective-C Class for Working with Fractions

Now it's time to define an actual class in Objective-C and learn how to work with instances of the class.

Once again, you'll learn procedure first. As a result, the actual program examples might not seem very practical. We get into more practical stuff later.

Suppose you need to write a program to work with fractions. Maybe you need to deal with adding, subtracting, multiplying, and so on. If you didn't know about classes, you might start with a simple program that looked like this:

Program 3.1

```
// Simple program to work with fractions

#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;
}
```

Program 3.1 **Output**

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

In Program 3.1 the fraction is represented in terms of its numerator and denominator. After the autorelease pool is created, the two lines in `main` both declare the variables `numerator` and `denominator` as integers and assign them initial values of `1` and `3`, respectively. This is equivalent to the following lines:

```
int numerator, denominator;

numerator = 1;
denominator = 3;
```

We represented the fraction `1/3` by storing `1` in the variable `numerator` and `3` in the variable `denominator`. If you needed to store a lot of fractions in your program, this could be cumbersome. Each time you wanted to refer to the fraction, you'd have to refer to the corresponding numerator and denominator. And performing operations on these fractions would be just as awkward.

It would be better if you could define a fraction as a single entity and collectively refer to its numerator and denominator with a single name, such as `myFraction`. You can do that in Objective-C, and it starts by defining a new class.

Program 3.2 duplicates the functionality of Program 3.1 using a new class called `Fraction`. Here, then, is the program, followed by a detailed explanation of how it works.

Program 3.2

```
// Program to work with fractions - class version

#import <Foundation/Foundation.h>

//---- @interface section ----

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

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

@end

//---- @implementation section ----


@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
   numerator = n;
}

-(void) setDenominator: (int) d
{
   denominator = d;
}

@end

//---- program section ----
```

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

    // Create an instance of a Fraction

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

    // Set fraction to 1/3

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

    // Display the fraction using the print method

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

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

Program 3.2 **Output**

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

As you can see from the comments in Program 3.2, the program is logically divided into three sections:

- @interface section
- @implementation section
- program section

The `@interface` section describes the class, its data components, and its methods, whereas the `@implementation` section contains the actual code that implements these methods. Finally, the `program` section contains the program code to carry out the intended purpose of the program.

Each of these sections is a part of every Objective-C program, even though you might not need to write each section yourself. As you'll see, each section is typically put in its own file. For now, however, we keep it all together in a single file.

The `@interface` Section

When you define a new class, you have to do a few things. First, you have to tell the Objective-C compiler where the class came from. That is, you have to name its *parent* class. Second, you have to specify what type of data is to be stored in the objects of this

class. That is, you have to describe the data that members of the class will contain. These members are called the *instance variables*. Finally, you need to define the type of operations, or *methods*, that can be used when working with objects from this class. This is all done in a special section of the program called the `@interface` section. The general format of this section looks like this:

```
@interface NewClassName: ParentClassName
{
    memberDeclarations;
}

methodDeclarations;
@end
```

By convention, class names begin with an uppercase letter, even though it's not required. This enables someone reading your program to distinguish class names from other types of variables by simply looking at the first character of the name. Let's take a short diversion to talk a little about forming names in Objective-C.

Choosing Names

In Chapter 2, "Programming in Objective-C," you used several variables to store integer values. For example, you used the variable `sum` in Program 2.4 to store the result of the addition of the two integers `50` and `25`.

The Objective-C language allows you to store data types other than just integers in variables as well, as long as the proper declaration for the variable is made before it is used in the program. Variables can be used to store floating-point numbers, characters, and even objects (or, more precisely, references to objects).

The rules for forming names are quite simple: They must begin with a letter or underscore (_), and they can be followed by any combination of letters (upper- or lowercase), underscores, or the digits `0`–`9`. The following is a list of valid names:

- `sum`
- `pieceFlag`
- `i`
- `myLocation`
- `numberOfMoves`
- `sysFlag`
- `ChessBoard`

On the other hand, the following names are not valid for the stated reasons:

- `sum$value $`—is not a valid character.
- `piece flag`—Embedded spaces are not permitted.

- `3Spencer`—Names can't start with a number.
- `int`—This is a reserved word.

`int` cannot be used as a variable name because its use has a special meaning to the Objective-C compiler. This use is known as a *reserved name* or *reserved word*. In general, any name that has special significance to the Objective-C compiler cannot be used as a variable name.

Always remember that upper- and lowercase letters are distinct in Objective-C. Therefore, the variable names `sum`, `Sum`, and `SUM` each refer to a different variable. As noted, when naming a class, start it with a capital letter. Instance variables, objects, and method names, on the other hand, typically begin with lowercase letters. To aid readability, capital letters are used inside names to indicate the start of a new word, as in the following examples:

- `AddressBook`—This could be a class name.
- `currentEntry`—This could be an object.
- `current_entry`—Some programmers use underscores as word separators.
- `addNewEntry`—This could be a method name.

When deciding on a name, keep one recommendation in mind: Don't be lazy. Pick names that reflect the intended use of the variable or object. The reasons are obvious. Just as with the comment statement, meaningful names can dramatically increase the readability of a program and will pay off in the debug and documentation phases. In fact, the documentation task will probably be much easier because the program will be more self-explanatory.

Here, again, is the `@interface` section from Program 3.2:

```
//---- @interface section ----

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

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

@end
```

The name of the new class is Fraction, and its parent class is `NSObject`. (We talk in greater detail about parent classes in Chapter 8, "Inheritance.") The `NSObject` class is

defined in the file NSObject.h, which is automatically included in your program whenever you import Foundation.h.

Instance Variables

The *memberDeclarations* section specifies what types of data are stored in a Fraction, along with the names of those data types. As you can see, this section is enclosed inside its own set of curly braces. For your Fraction class, these declarations say that a Fraction object has two integer members, called numerator and denominator:

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

The members declared in this section are known as the instance variables. As you'll see, each time you create a new object, a new and unique set of instance variables also is created. Therefore, if you have two Fractions, one called fracA and another called fracB, each will have its own set of instance variables. That is, fracA and fracB each will have its own separate numerator and denominator. The Objective-C system automatically keeps track of this for you, which is one of the nicer things about working with objects.

Class and Instance Methods

You have to define methods to work with your Fractions. You need to be able to set the value of a fraction to a particular value. Because you won't have direct access to the internal representation of a fraction (in other words, direct access to its instance variables), you must write methods to set the numerator and denominator. You'll also write a method called print that will display the value of a fraction. Here's what the declaration for the print method looks like in the interface file:

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

The leading minus sign (-) tells the Objective-C compiler that the method is an instance method. The only other option is a plus sign (+), which indicates a class method. A class method is one that performs some operation on the class itself, such as creating a new instance of the class.

An instance method performs some operation on a particular instance of a class, such as setting its value, retrieving its value, displaying its value, and so on. Referring to the car example, after you have manufactured the car, you might need to fill it with gas. The operation of filling it with gas is performed on a particular car, so it is analogous to an instance method.

Return Values

When you declare a new method, you have to tell the Objective-C compiler whether the method returns a value and, if it does, what type of value it returns. You do this by enclosing the return type in parentheses after the leading minus or plus sign. So this declaration specifies that the instance method called `currentAge` returns an integer value:

```
-(int) currentAge;
```

Similarly, this line declares a method that returns a double precision value. (You'll learn more about this data type in Chapter 4, "Data Types and Expressions.")

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

A value is returned from a method using the Objective-C `return` statement, similar to the way in which we returned a value from `main` in previous program examples.

If the method returns no value, you indicate that using the type `void`, as in the following:

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

This declares an instance method called `print` that returns no value. In such a case, you do not need to execute a `return` statement at the end of your method. Alternatively, you can execute a `return` without any specified value, as in the following:

```
return;
```

Method Arguments

Two other methods are declared in the `@interface` section from Program 3.2:

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

These are both instance methods that return no value. Each method takes an integer argument, which is indicated by the `(int)` in front of the argument name. In the case of `setNumerator`, the name of the argument is `n`. This name is arbitrary and is the name the method uses to refer to the argument. Therefore, the declaration of `setNumerator` specifies that one integer argument, called `n`, will be passed to the method and that no value will be returned. This is similar for `setDenominator`, except that the name of its argument is `d`.

Notice the syntax of the declaration for these methods. Each method name ends with a colon, which tells the Objective-C compiler that the method expects to see an argument. Next, the type of the argument is specified, enclosed in a set of parentheses, in much the same way the return type is specified for the method itself. Finally, the symbolic name to be used to identify that argument in the method is specified. The entire declaration is terminated with a semicolon. Figure 3.1 depicts this syntax.

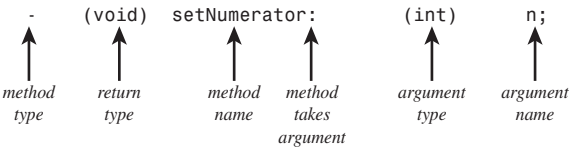

Figure 3.1 Declaring a method

When a method takes an argument, you also append a colon to the method name when referring to the method. Therefore, `setNumerator:` and `setDenominator:` is the correct way to identify these two methods, each of which takes a single argument. Also, identifying the `print` method without a trailing colon indicates that this method does not take any arguments. In Chapter 7, "More on Classes," you'll see how methods that take more than one argument are identified.

The `@implementation` Section

As noted, the `@implementation` section contains the actual code for the methods you declared in the `@interface` section. Just as a point of terminology, you say that you declare the methods in the `@interface` section and that you *define* them (that is, give the actual code) in the `@implementation` section.

The general format for the `@implementation` section is as follows:

```
@implementation NewClassName
  methodDefinitions;
@end
```

`NewClassName` is the same name that was used for the class in the `@interface` section. You can use the trailing colon followed by the parent class name, as we did in the `@interface` section:

```
@implementation Fraction: NSObject
```

However, this is optional and typically not done.

The *`methodDefinitions`* part of the `@implementation` section contains the code for each method specified in the `@interface` section. Similar to the `@interface` section, each method's definition starts by identifying the type of method (class or instance), its return type, and its arguments and their types. However, instead of the line ending with a semicolon, the code for the method follows, enclosed inside a set of curly braces.

Consider the `@implementation` section from Program 3.2:

```
//---- @implementation section ----
@implementation Fraction
-(void) print
{
    NSLog (@"%i/%i", numerator, denominator);
}
```

```
-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end
```

The `print` method uses `NSLog` to display the values of the instance variables `numerator` and `denominator`. But to which numerator and denominator does this method refer? It refers to the instance variables contained in the object that is the receiver of the message. That's an important concept, and we return to it shortly.

The `setNumerator:` method stores the integer argument you called `n` in the instance variable `numerator`. Similarly, `setDenominator:` stores the value of its argument `d` in the instance variable `denominator`.

The `program` Section

The `program` section contains the code to solve your particular problem, which can be spread out across many files, if necessary. Somewhere you must have a routine called `main`, as we've previously noted. That's where your program always begins execution. Here's the program section from Program 3.2:

```
//---- program section ----

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

   Fraction  *myFraction;

   // Create an instance of a Fraction and initialize it

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

   // Set fraction to 1/3

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

```
    // Display the fraction using the print method

    NSLog (@"The value of myFraction is:");
    [myFraction print];

    [myFraction release];
    [pool drain];

    return 0;
}
```

Inside `main`, you define a variable called `myFraction` with the following line:

```
Fraction *myFraction;
```

This line says that `myFraction` is an object of type `Fraction`; that is, `myFraction` is used to store values from your new `Fraction` class. The asterisk that precedes the variable name is described in more detail below.

Now that you have an object to store a `Fraction`, you need to create one, just as you ask the factory to build you a new car. This is done with the following line:

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

`alloc` is short for *allocate*. You want to allocate memory storage space for a new fraction. This expression sends a message to your newly created `Fraction` class:

```
[Fraction alloc]
```

You are asking the `Fraction` class to apply the `alloc` method, but you never defined an `alloc` method, so where did it come from? The method was inherited from a parent class. Chapter 8, "Inheritance" deals with this topic in detail.

When you send the `alloc` message to a class, you get back a new instance of that class. In Program 3.2, the returned value is stored inside your variable `myFraction`. The `alloc` method is guaranteed to zero out all of an object's instance variables. However, that doesn't mean that the object has been properly initialized for use. You need to initialize an object after you allocate it.

This is done with the next statement in Program 3.2, which reads as follows:

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

Again, you are using a method here that you didn't write yourself. The `init` method initializes the instance of a class. Note that you are sending the `init` message to `myFraction`. That is, you want to initialize a specific `Fraction` object here, so you don't send it to the class—you send it to an instance of the class. Make sure you understand this point before continuing.

The `init` method also returns a value—namely, the initialized object. You store the return value in your `Fraction` variable `myFraction`.

The two-line sequence of allocating a new instance of class and then initializing it is done so often in Objective-C that the two messages are typically combined, as follows:

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

This inner message expression is evaluated first:

```
[Fraction alloc]
```

As you know, the result of this message expression is the actual `Fraction` that is allocated. Instead of storing the result of the allocation in a variable, as you did before, you directly apply the `init` method to it. So, again, first you allocate a new `Fraction` and then you initialize it. The result of the initialization is then assigned to the `myFraction` variable.

As a final shorthand technique, the allocation and initialization is often incorporated directly into the declaration line, as in the following:

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

We use this coding style often throughout the remainder of this book, so it's important that you understand it. You've seen in every program up to this point with the allocation of the autorelease pool:

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

Here an `alloc` message is sent to the `NSAutoreleasePool` class requesting that a new instance be created. The `init` message then is sent to the newly created object to get it initialized.

Returning to Program 3.2, you are now ready to set the value of your fraction. These program lines do just that:

```
// Set fraction to 1/3

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

The first message statement sends the `setNumerator:` message to `myFraction`. The argument that is supplied is the value `1`. Control is then sent to the `setNumerator:` method you defined for your `Fraction` class. The Objective-C system knows that it is the method from this class to use because it knows that `myFraction` is an object from the `Fraction` class.

Inside the `setNumerator:` method, the passed value of `1` is stored inside the variable `n`. The single program line in that method stores that value in the instance variable `numerator`. So you have effectively set the numerator of `myFraction` to `1`.

The message that invokes the `setDenominator:` method on `myFraction` follows next. The argument of `3` is assigned to the variable `d` inside the `setDenominator:` method. This value is then stored inside the `denominator` instance variable, thus completing the

assignment of the value 1/3 to myFraction. Now you're ready to display the value of your fraction, which you do with the following lines of code from Program 3.2:

```
// Display the fraction using the print method

NSLog (@"The value of myFraction is:");
[myFraction print];
```

The NSLog call simply displays the following text:

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

The following message expression invokes the print method:

```
[myFraction print];
```

Inside the print method, the values of the instance variables numerator and denominator are displayed, separated by a slash character.

The message in the program releases or frees the memory that was used for the Fraction object:

```
[myFraction release];
```

This is a critical part of good programming style. Whenever you create a new object, you are asking for memory to be allocated for that object. Also, when you're done with the object, you are responsible for releasing the memory it uses. Although it's true that the memory will be released when your program terminates anyway, after you start developing more sophisticated applications, you can end up working with hundreds (or thousands) of objects that consume a lot of memory. Waiting for the program to terminate for the memory to be released is wasteful of memory, can slow your program's execution, and is not good programming style. So get into the habit of releasing memory when you can right now.

The Apple runtime system provides a mechanism known as *garbage collection* that facilitates automatic cleanup of memory. However, it's best to learn how to manage your memory usage yourself instead of relying on this automated mechanism. In fact, you can't rely on garbage collection when programming for certain platforms on which garbage collection is not supported, such as the iPhone or iPad. For that reason, we don't talk about garbage collection until much later in this book.

It seems as if you had to write a lot more code to duplicate in Program 3.2 what you did in Program 3.1. That's true for this simple example here; however, the ultimate goal in working with objects is to make your programs easier to write, maintain, and extend. You'll realize that later.

Let's go back for a second to the declaration of myFraction

```
Fraction *myFraction;
```

and the subsequent setting of its values.

> **Note**
>
> Note that Xcode inserts a space after the * when it generates the first line in `main` that creates an `NSAutoreleasepool` object. It's not needed, so we won't do that for our objects, as most programmers don't either.

The asterisk (*) in front of `myFraction` in its declaration says that `myFraction` is actually a reference (or *pointer*) to a `Fraction` object. The variable `myFraction` doesn't actually store the fraction's data (that is, its numerator and denominator values). Instead, it stores a reference—which is a actually a memory address—indicating where the object's data is located in memory. When you first declare `myFraction` as shown, its value is undefined as it has not been set to any value and does not have a default value. We can conceptualize `myFraction` as a box that holds a value. Initially the box contains some undefined value, as it hasn't been assigned any value. This is depicted in Figure 3.2.

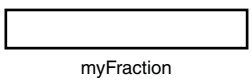

Figure 3.2 Declaring
`Fraction *myFraction;`

When you allocate a new object (using `alloc`, for example) enough space is reserved in memory to store the object's data, which inclues space for its instance variables, plus a little more. The location of where that data is stored (the reference to the data) is returned by the `alloc` routine, and assigned to the variable `myFraction`. This all takes place when this statement in executed in Program 3.2.

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

The allocation of the object and the storage of the reference to that object in `myFraction` is depicted in Figure 3.3:

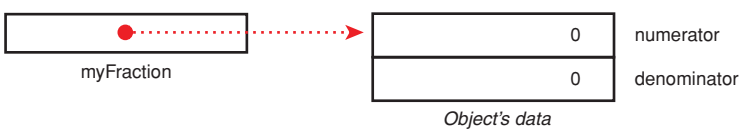

Figure 3.3 Relationship between `myFraction` and its data

> **Note**
>
> There's some more data stored with the object than that indicated, but you don't need to worry about that here. You'll note that the instance variables are shown as being set to 0. That's currently being handled by the `alloc` method. However, the object still has not been properly initialized. You still need to use the `init` method on the newly allocated object.

Notice the directed line in Figure 3.3. This indicates the connection that has been made between the variable `myFraction` and the allocated object. (The value stored inside `myFraction` is actually a memory address. It's at that memory address that the object's data is stored.)

Subsequently in Program 3.2, the fraction's numerator and denominator are set. Figure 3.4 depicts the fully initialized `Fraction` object with its numerator set to 1 and its denominator set to 3.

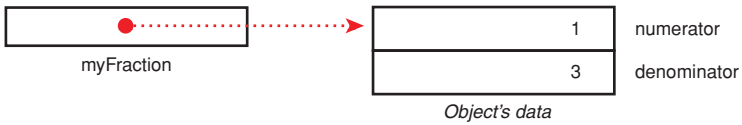

Figure 3.4 Setting the fraction's numerator and denominator

The next example shows how you can work with more than one fraction in your program. In Program 3.3, you set one fraction to `2/3`, set another to `3/7`, and display them both.

Program 3.3

```
// Program to work with fractions - cont'd

#import <Foundation/Foundation.h>

//---- @interface section ----

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

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

@end

//---- @implementation section ----


@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
   numerator = n;
}

-(void) setDenominator: (int) d
{
   denominator = d;
}
```

```
@end

//---- program section ----

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

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

   // Set 1st fraction to 2/3

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

   // Set 2nd fraction to 3/7

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

   // Display the fractions

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

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

   [frac1 release];
   [frac2 release];

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

Program 3.3 **Output**

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

The `@interface` and `@implementation` sections remain unchanged from Program 3.2. The program creates two `Fraction` objects, called `frac1` and `frac2`, and then assigns the value `2/3` to the first fraction and `3/7` to the second. Realize that when the `setNumerator:` method is applied to `frac1` to set its numerator to `2`, the instance variable `frac1` gets its instance variable `numerator` set to `2`. Also, when `frac2` uses the same method to set its numerator to `3`, its distinct instance variable `numerator` is set to the value `3`. Each time you create a new object, it gets its own distinct set of instance variables. Figure 3.5 depicts this.

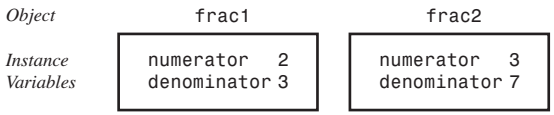

Figure 3.5 Unique instance variables

Based on which object is getting sent the message, the correct instance variables are referenced. Therefore, here `frac1`'s `numerator` is referenced whenever `setNumerator:` uses the name `numerator` inside the method:

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

That's because `frac1` is the receiver of the message.

Accessing Instance Variables and Data Encapsulation

You've seen how the methods that deal with fractions can access the two instance variables `numerator` and `denominator` directly by name. In fact, an instance method can always directly access its instance variables. A class method can't, however, because it's dealing only with the class itself, not with any instances of the class (think about that for a second). But what if you wanted to access your instance variables from someplace else—for example, from inside your `main` routine? You can't do that directly because they are hidden. The fact that they are hidden from you is a key concept called *data encapsulation*. It enables someone writing class definitions to extend and modify the class definitions, without worrying about whether programmers (that is, users of the class) are tinkering with the internal details of the class. Data encapsulation provides a nice layer of insulation between the programmer and the class developer.

You can access your instance variables in a clean way by writing special methods to set and retrieve their values. We wrote `setNumerator:` and `setDenominator:` methods to set the values of the two instance variables in our `Fraction` class. To retrieve the values of those instance variables, you'll need to write two new methods. For example, you'll create two new methods called, appropriately enough, `numerator` and `denominator` to access the corresponding instance variables of the `Fraction` that is the receiver of the message. The result is the corresponding integer value, which you return. Here are the declarations for your two new methods:

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

And here are the definitions:

```
-(int) numerator
{
```

```
   return numerator;
}

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

Note that the names of the methods and the instance variables they access are the same. There's no problem doing this (although it might seem a little odd at first); in fact, it is common practice. Program 3.4 tests your two new methods.

Program 3.4

```
// Program to access instance variables - cont'd

#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

//---- @implementation section ----


@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


//---- program section ----

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

    // Set fraction to 1/3

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

    // Display the fraction using our two new methods

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

    return 0;
}
```

Program 3.4 **Output**

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

This `NSLog` statement displays the results of sending two messages to `myFraction:` the first to retrieve the value of its `numerator`, and the second the value of its `denominator`:

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

So, in the first message call, the `numerator` message will be sent to the `Fraction` object `myFraction`. In that method, the code will return the value of the `numerator` instance variable for that fraction. Remember, the context of a method while it is executing is the object that is the receiver of the message. So when the `numerator` method accesses and returns the value of the `numerator` instance variable, it's `myFraction`'s numerator that will be accessed and returned. That returned integer value is then passed along to `NSLog` to be displayed.

For the second message call, the `denominator` method will be called to access and return the value of `myFraction`'s denominator, which is then passed to `NSLog` to be displayed.

Incidentally, methods that set the values of instance variables are often collectively referred to as *setters*, and methods used to retrieve the values of instance variables are called *getters*. For the `Fraction` class, `setNumerator:` and `setDenominator:` are the setters, and `numerator` and `denominator` are the getters. Collectively, setters and getters are also referred to as *accessor* methods.

> **Note**
> Soon you'll learn a convenient feature of Objective-C 2.0 that allows for the automatic creation of getter and setter methods.

Make sure you understand the difference between setters and the getters. The setters don't return a value because their purpose is to take an argument and to set the corresponding instance variable to the value of that argument. No value needs to be returned in that case. That's the purpose of a setter: to set the value of an instance variable, so setters typically do not return values. On the other hand, the purpose of the getter is to "get" the value of an instance variable stored in an object and to send it back to the program. In order to do that, the getter must return the value of the instance variable using the `return` statement.

Again, the idea that you can't directly set or get the value of an instance variable from outside of the methods written for the class, but instead have to write setter and getter methods to do so is the principle of data encapsulation. So you have to use methods to access this data that is normally hidden to the "outside world." This provides a centralized path to the instance variables and prevents some other code from indirectly changing these values, which would make your programs harder to follow, debug, and modify.

We should also point out that there's also a method called `new` that combines the actions of an `alloc` and `init`. So this line could be used to allocate and initialize a new `Fraction`:

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

It's generally better to use the two-step allocation and initialization approach so you conceptually understand that two distinct events are occurring: You're first creating a new object and then you're initializing it.

Summary

Now you know how to define your own class, create objects or instances of that class, and send messages to those objects. We return to the `Fraction` class in later chapters. You'll learn how to pass multiple arguments to your methods, how to divide your class definitions into separate files, and also how to use key concepts such as inheritance and dynamic binding. However, now it's time to learn more about data types and writing expressions in Objective-C. First, try the exercises that follow to test your understanding of the important points covered in this chapter.

Exercises

1. Which of the following are invalid names? Why?

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

2. Based on the example of the car in this chapter, think of an object you use every day. Identify a class for that object and write five actions you do with that object.

3. Given the list in exercise 2, use the following syntax to rewrite your list in this format:

```
[instance method];
```

4. Imagine that you owned a boat and a motorcycle in addition to a car. List the actions you would perform with each of these. Do you have any overlap between these actions?

5. Based on question 4, imagine that you had a class called `Vehicle` and an object called `myVehicle` that could be either `Car`, `Motorcycle`, or `Boat`. Imagine that you wrote the following:

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

   Do you see any advantages of being able to apply an action to an object that could be from one of several classes?

6. In a procedural language such as C, you think about actions and then write code to perform the action on various objects. Referring to the car example, you might write a procedure in C to wash a vehicle and then inside that procedure write code to handle washing a car, washing a boat, washing a motorcycle, and so on. If you took that approach and then wanted to add a new vehicle type (see the previous exercise), do you see advantages or disadvantages to using this procedural approach over an object-oriented approach?

7. Define a class called `XYPoint` that will hold a Cartesian coordinate (x, y), where x and y are integers. Define methods to individually set the x and y coordinates of a point and retrieve their values. Write an Objective-C program to implement your new class and test it.

This page intentionally left blank

Index

SYMBOLS

A

B

C

D

E

F

G

H

I–J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z