Jesse Feiler

In Full Color

SECOND EDITION

Figures and code appear as they do in Xcode 5

Sams Teach Yourself

Objective-C

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Jesse Feiler

Sams **Teach Yourself**

Objective-C

in 24 Hours

Second Edition

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Objective-C in 24 Hours, Second Edition

ISBN-13: 978-0-672-33449-8
ISBN-10: 0-672-33449-6

Library of Congress Control Number: 2013954664

Printed in the United States of America

First Printing March 2014

Trademarks

Warning and Disclaimer

Special Sales

Editor-in-Chief
Greg Wiegand

Executive Editor
Loretta Yates

Development Editor
Sondra Scott

Marketing Manager
Stephane Nakib

Managing Editor
Kristy Hart

Senior Project Editor
Lori Lyons

Copy Editor
Karen Annett

Indexer
Erika Millen

Proofreader
Kathy Ruiz

Technical Editor
Robert McGovern

Publishing Coordinator
Cindy Teeters

Cover Designer
Mark Shirar

Compositor
Nonie Ratcliff

Manufacturing Buyer
Dan Uhrig

Contents at a Glance

Part IV: Beyond the Basics

Part V: Appendixes

Table of Contents

Part II: Working with the Objective-C Basics

Part IV: Beyond the Basics

Part V: Appendixes

About the Author

Jesse Feiler is a developer and author. He has been an Apple developer since before it became fashionable, and has worked with mobile devices starting with Apple's Newton and continuing with the iOS products (iPhone, iPod touch, and iPad).

His books include *Sams Teach Yourself Core Data in 24 Hours*, *Data-Driven iOS Apps for iPad and iPhone with FileMaker Pro*, *FileMaker Pro in Depth* (Sams/Pearson), *Database-Driven Web Sites* (Harcourt), *Learning iCloud Data Management* (Addison Wesley/Pearson), and *iOS App Development for Dummies* (Wiley).

He has written about Objective-C and the Apple frameworks beginning with *Rhapsody Developer's Guide* (AP Professional) and *Mac OS X Developer's Guide* (Morgan Kaufmann).

He is the author of *MinutesMachine*, the meeting management software for iPad, as well as Saranac River Trail app for iPhone and iPad. There are more details at champlainarts.com.

A native of Washington DC, he has lived in New York City and currently lives in Plattsburgh, NY. He can be reached at northcountryconsulting.com.

Acknowledgments

As always, Carole Jelen at Waterside Productions has provided help and guidance in bringing this book to fruition. At Pearson, Loretta Yates has helped move this book from an idea to an actual book. Along the way, the book and author have benefited from technical suggestions from Robert McGovern. The production staff kept the book on track and helped clarify text.

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.

We welcome your comments. You can email or write 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 better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

Introduction

If you want to develop native apps for OS X or for iOS, the language in which you write them is Objective-C. (You can write web-based apps using Safari extensions and HTML5.)

Because Apple's development environment builds on the frameworks of Cocoa and Cocoa Touch, what you have to write is the code that is specific to your own app. All the basic functionality is provided by Cocoa and the frameworks; you do not have to write any code to manage menus, for example (in most cases), and you don't have to worry about implementing complex view structures on iOS such as navigation bars and split views. The code that you write is very specific code that fits into the existing frameworks. It might be code that overrides or extends an existing method, or it might be new methods that are unique to your own app.

Who Should Read This Book

Even though the code that you write might not be extensive, you do have to understand the syntax and structure of Objective-C in order to write it and to understand the code that Apple provides for you.

You should have some basic experience with programming in order to read this book, but it is designed to bring you up to speed on the major prerequisites to understanding Objective-C, including the concepts of object-oriented programming. The book assumes that you have at least a basic understanding of the C programming language on which Objective-C is based. Don't worry if you are far from a C expert; the concepts used in Objective-C are the basics.

What This Book Covers

Objective-C has been the language of choice for the Cocoa environments and, before them, the NeXTSTEP and Rhapsody environments. Theoretically, it can be used in other contexts, but in practice, it is the language of these environments. Sometimes, drawing the line between Objective-C syntax and elements of the Cocoa environments and frameworks can be very difficult. This book focuses on the language itself. It provides a number of examples from code in

the Cocoa frameworks as well as code in the examples from developer.apple.com and the templates built in to Xcode, but the primary focus is on the language. The book includes information on how to use the Xcode development tool because, for all intents and purposes, you have to use it to write Objective-C code.

Downloading the Sample Files

You can download examples from the author's website at northcountryconsulting.com or from the publisher's site at informit.com/title/9780672334498.

How This Book Is Organized

There are five parts to this book. You can focus on whichever one addresses an immediate problem, or you can get a good overview by reading straight through. As with all of the Sams Teach Yourself books, as much as possible each hour is made to stand on its own so that you can jump around to learn in your own way. Cross-references throughout the book help you find related material.

Part I: Getting Started with Objective-C

This part gives you the high-level view of Objective-C.

- **Hour 1, "Overview of the Developer Program**—Here you see how to register to become an Apple developer and gain access to the resources on developer.apple.com. You can choose from various options, but you must register before you can do any meaningful development work.
- **Hour 2, "Object-Oriented Programming with Objective-C"**—This hour shows you how Objective-C builds on top of C. It also covers how Objective-C implements object-oriented concepts and manages inheritance.
- **Hour 3, "Using Object-Oriented Features in Objective-C"**—Objective-C is a dynamic language that relies on messaging. Those concepts are explained here so that you can begin to implement your own apps.
- **Hour 4, "Using Xcode 5"**— This is the integrated development environment (IDE) you use to develop apps. You'll see how to work with new features that make your coding faster and catches errors even before you build your app. Xcode integrates the Git source code repository so that you can easily track your code on your own Mac, a networked Mac, or a public repository such as GitHub.
- **Hour 5, "Using Compiler Directives"**—Compiler directives help you manage files and use the preprocessor to prepare code for the compiler itself.

Part II: Working with the Objective-C Basics

This part of the book delves into the concepts of messaging, classes, selectors, building blocks, and memory.

- **Hour 6, "Exploring Messaging and a Testbed App"**—Messaging is the heart of Objective-C. You send messages rather than call methods or functions, and the difference between Objective-C and other languages is significant.
- **Hour 7, "Declaring a Class in an Interface File"**—This hour explores the basics of declaring a class. You see how to import other files and use forward declarations for classes and protocols.
- **Hour 8, "Declaring Instance Variables in an Interface File"**—You can assign instance variables to classes. You also see how to deal with static typing and the scope of instance variables.
- **Hour 9, "Declaring Properties in an Interface File"**—Objective-C lets you declare properties of a class in your interface file rather than declaring variables. The property declaration provides more features than a simple C variable declaration. Furthermore, the compiler can synthesize the accessors for the property based on the property declaration.
- **Hour 10, "Declaring Methods in an Interface File"**—Method declarations in Objective-C provide the same functionality that they do in other languages, but the details in this message-based environment are different.
- **Hour 11, "Declaring Actions in an Interface File"**—Actions are typically triggered by user actions in the interface. You declare them in your interface files and manage their interactions with Interface Builder.
- **Hour 12, "Routing Messages with Selectors"**—The target-action pattern is the basis for message routing; it is explained in this hour. You also see how to use selectors and the SEL data type.
- **Hour 13, "Building on the Foundation"**—Throughout Objective-C, you find certain patterns repeated over and over. These include its data types and the concept of mutable and immutable versions of them.
- **Hour 14, "Defining a Class in an Implementation File"**—After all those declarations, you get down to actually defining the code behind the declarations in this hour.
- **Hour 15, "Organizing Data with Collections"**—Arrays are one of the core features of programming languages. Objective-C adds additional collection objects, including dictionaries and sets. This hour covers how to use them.

- **Hour 16, "Managing Memory and Runtime Objects"**—Whether you are running on a mainframe or supercomputer or a relatively small mobile device, memory is a critical resource. There are many tools to help you manage memory, but in most cases, you still need to do something and understand what your choices are. This hour introduces you to Automatic Reference Counting (ARC), the modern way to manage Objective-C memory; it also shows you how garbage collection has been used in the past.

Part III: Expanding and Extending Classes

One of the differences between Objective-C and other object-oriented languages is that you can expand and extend classes in a variety of ways and not just by subclassing them.

- **Hour 17, "Extending a Class with Protocols and Delegates"**—Next to the ability to override classes, protocols and delegates are the most commonly used way for adding functionality to a variety of classes.
- **Hour 18, "Extending a Class with Categories and Extensions"**—Categories and extensions let you add methods and even variables to existing classes—even those for which you might not have the source code.
- **Hour 19, "Using Associative References and Fast Enumeration"**—These relatively recent additions to Objective-C enable you to become a power programmer. You can add and remove associations at runtime, which gives you more flexibility with your instance variables. Fast enumeration also speeds your runtime performance and might mean that you have less code to write yourself.
- **Hour 20, "Working with Blocks"**—Blocks provide the ability to create and use sections of code that are portable and anonymous. You can use them as arguments of methods or functions, typically as completion handlers for completion routines, handlers, and errors, as well as for specific types of functionality, such as sorting and view manipulation. The concept is borrowed from languages such as C, Ruby, Python, and Lisp; it has been part of Objective-C since Mac OS X 10.6 (Snow Leopard) and iOS 4.0.

Part IV: Beyond the Basics

Exception and error handling, queues and threading, the debugger, and Xcode debug gauges allow you to analyze and improve your projects and their performance.

- **Hour 21, "Handling Exceptions and Errors"**—As in many languages, you can set up try/catch/finally structures to catch exceptions as close as possible to the place where they happen. In this hour, you see how to use them and how to integrate them with the debugger.

- **Hour 22, "Grand Central Dispatch: Using Queues and Threading"**—Even on small mobile devices, you have the ability to manage multiprocessing. This hour shows you the techniques that are available for multiprocessing.
- **Hour 23, "Working with the Debugger"**—During development with Xcode, you can use the debugger and console to improve your code. This hour shows you the basics and explores break points, preferences, and some advanced debugging techniques.
- **Hour 24, "Using Xcode Debug Gauges for Analysis"**—Although debugging can help you get rid of bugs, you need to monitor your app's performance. Memory leaks aren't bugs unless they crash your app, but with these tools you can spot them before the damage is done. Debug gauges are a simple graphical interface on top of the more sophisticated Instruments app; both are discussed here.

Part V: Appendixes

The appendixes provide additional reference material that can help you with the concepts in the book. There is even more material on developer.apple.com and on the author's website at northcountryconsulting.com.

- **Appendix A, "C Syntax Summary"**—Objective-C is a thin layer built on top of C. This appendix reviews the basic C syntax that matters for Objective-C. Much of it is reimplemented for you in the frameworks of Cocoa, so this appendix walks you through what is left.
- **Appendix B, "Apps, Packages, and Bundles"**—Your app on Mac OS X or iOS is actually a collection of files, including code, resources such as data stores and images, and property lists. This is a high-level overview so that you can understand what it is you are building with Xcode and Objective-C.
- **Appendix C, "Archiving and Packaging Apps for Development and Testing"**—This overview gives you the basic information you need to know to share your apps with testers and through the App Store. It is not just a matter of copying files.

This page intentionally left blank

HOUR 3
Using Object-Oriented Features in Objective-C

What You'll Learn in This Hour

- Sending and receiving messages
- Looking inside a message
- Initializing objects

Communicating to Methods with Messages

Perhaps the biggest difference between Objective-C and languages such as C++ is its messaging syntax as well as the way people talk about it. Objective-C has classes just as other object-oriented languages do, and those classes can have methods within them. You communicate with those methods with messages. A message is enclosed within square brackets, and it consists of the name of the object to which it is being sent followed by the message itself.

The implementation files that you create carry the .m suffix because originally they were referred to as message files that contained the code for the messages defined in the header (.h) files. (This is possibly an apocryphal tale, but the importance of messaging in Objective-C is undisputed.)

NOTE

There is an ulterior purpose to this section. It does help you to understand how to communicate with Objective-C methods and messages, but the examples begin with the messages that you use to allocate and initialize objects.

Looking at a Simple Message

Here is a simple message that is sent to an object called `myObject`, which is assumed to be of type `NSObject`—the object that is the root class of most class hierarchies in Objective-C.

```
[myObject init];
```

This message calls the init method on `myObject`.

NOTE

That is the point made previously about the different way many people talk about Objective-C methods: They often refer to calling a message "on" an object.

Methods can return a value. If a method returns a value, you can set it to a variable as in the following:

```
myVariable = [myObject init];
```

Declaring a Method

When you declare a simple method, you use an Objective-C variation on C function syntax. `NSObject`, the root class of almost all of the Objective-C objects you use, does declare an `init` method.

NOTE

`NSObject` Class

As a matter of simplification, `NSObject` is referred to as the superclass of all objects in the Objective-C frameworks for Cocoa and Cocoa Touch. Note the NS prefix, which indicates that this method dates back to NeXTSTEP. Although it does not matter in most cases, the fact that `NSObject` is only the superclass of objects in the frameworks (and not all Objective-C objects) is something to bear in mind.

The following is the declaration that supports the messages shown in the previous section:

```
- (id)init
```

As you might surmise, the `init` method shown here returns a result of type `id`. (You find out more about `id` shortly.)

The minus sign at the start of the method is an important part of the declaration: It is the method type. It indicates that this is a method that is defined for instances of a class. Any instance of the class in which this declaration is used can invoke this method.

To put it another way, you (or an instance of a class) can send the `init` message to any instance of this class. Because this is the `NSObject` superclass of every other object, that means you can send the `init` message to any instance of any class.

There is more on allocating and initializing objects later in this hour.

Using Class Methods

The minus sign at the start of the method shown in the previous section indicates that it is an instance method. There is another type of method in Objective-C: a class method. It is indicated by a plus sign.

TIP

Class objects are often used as *factory objects* to create new, concrete instances of themselves.

A message to an instance method can be sent to any instance of that class subject to constraints for that specific class. Whereas you call an instance method on an instance of a class, you call a class method on the class itself. No instance is involved.

Class methods are used most frequently as *factory methods*. Perhaps the most common class method is `alloc`. For `NSObject`, its declaration is

```
+ (id)alloc;
```

Whereas you send `init` to an instance, as in this case:

```
[myObject init];
```

`alloc` allocates an uninitialized instance of a class as in

```
[MyClass alloc];
```

This returns an instance of `MyClass`. As you can see in the declaration, this result is of type `id`. It is time to explore that type.

Working with `id`—Strongly and Weakly Typed Variables

Objective-C supports strongly and weakly typed variables. When you reference a variable using a strong type, you specify the type of the variable. The actual variable must be of that type or a subclass of that type; if it is a subclass, it is, by definition, the type of all of its superclasses.

In Cocoa, you can declare a variable as:

```
NSArray *myArray
```

This means you could be referring to an object of type `NSMutableArray`, which is a subclass. You can write the same code to work with elements of the array no matter what its actual type is. If necessary, you might have to coerce a specific instance to the subclass that you want (if you know that is what it is).

`id` is the ultimate weakly typed variable; it could be any class. That is why it is used as the return type from `alloc`. `alloc` is a class method on `NSObject` so if you call it on an `NSArray`, you get an instance of `NSArray` returned through `id`.

NOTE

instanceType

A new keyword, `instanceType`, is available when working with related result types. By convention, methods beginning with names such as `init` and `alloc` always return objects that are an instance of the class type that receives the message. Thus, `[MyClass init]` returns an instance of `MyClass`. `Init` is declared as returning an `id`, but with this convention it is the related type. You can now use `instanceType` instead of `id` to specify that the returned value is the same type as the receiver's type (that is, `MyClass` in this example). Because `instanceType` turns out to be more specific than `id` at compile time, it is preferable to use in these cases because it can allow for more error checking. As you can see from the examples in this hour, many types of the code you write already use specific variables of the correct type; however, `instanceType` in declarations of your own methods can make that more likely.

Nesting Messages

Messages can be nested within one another. You could write the following:

```
myObject = [MyClass alloc];
myObject = [myObject init];
```

This would use the class method of `MyClass` to allocate a new instance of `MyClass`, which you immediately assign to `myObject`.

You can nest them together as follows:

```
myObject = [[MyClass alloc] init];
```

The rules for nesting square brackets are the same as for nesting parentheses——the innermost set is evaluated first.

TIP

Xcode gives you a big assist by enabling you to set a preference to match brackets as you type.

Looking at Method Signatures and Parameters

`alloc` and `init` are very good basic examples because they have no parameters. Most methods in any language do have parameters. For example, you can write an area function that takes two parameters (height and width) and returns the product as its return value.

Other languages generally specify a name and a type for each parameter, and so does Objective-C. However, it adds another dimension: It labels each parameter.

This labeling means that the code is more readable, but you do have to understand what is going on when there is more than one parameter. When there is no parameter, the message is simply the receiver and the name of the method:

```
[myObject init];
```

If there is a parameter, it follows the method name. In the message, a colon precedes the parameter itself. For example, in `NSSet`, you can initialize a set with an `NSArray` using code like this:

```
mySet =[NSSet alloc];
[mySet initWithArray: myArray];
```

The declaration needs to specify not only the parameter name, which is used in the code of the method, but also its type:

```
(instanceType)initWithArray:(NSArray *)array;
```

The second and subsequent parameters are also labeled. The difference is that the first parameter is labeled in effect by the name of the method. If you add more parameters, their names and types are needed; they are preceded by a keyword (which, in the case of the first parameter is the method name). Here is another `NSSet` method. It initializes an `NSSet` to the elements of another `NSSet` (the first parameter). The second parameter specifies whether the elements of the first set are to be copied or not.

Here is a typical invocation:

```
[mySet: initWithSet: aSet copyItems:YES];
```

Here is the declaration:

```
(instanceType)initWithSet:(NSSet *)set copyItems:(BOOL)flag
```

In documentation (and in this book), the signature sometimes is compressed to be the result, method name, and parameters so that the previous declaration is shown as

```
(instanceType)initWithSet:copyItems:
```

Allocating and Initializing Objects

The messages shown in this hour have demonstrated how you can allocate and initialize objects. Because these processes are used so extensively in Objective-C, it is worthwhile to take a few moments to look more carefully at these messages.

As noted previously, you most often create instances of classes by calling `alloc`. This is a class method of `NSObject`, which means it is available to any class in the frameworks or in your own code. You simply send a message to the class, and you get back a new instance.

```
MyClass *newInstance = [MyClass alloc];
```

Immediately thereafter, you initialize the newly created instance, most often with a call to `init` or a related method.

```
newInstance = [newInstance init];
```

As noted previously, you can combine these two messages into one line of code.

```
MyClass *newInstance = [[MyClass alloc] init];
```

It makes sense that `alloc` returns an object that you can place in an instance variable. You might be surprised that `init` also returns an object. The reason for this is that after you have created the new object, the call to `init` might not only set instance variables and other settings, but it also might decide to replace the allocated object with another one. That is the one that you should use thereafter.

By convention, initializer names start with `init`. They might have additional parameters, such as these various initializers for `UIBarButtonItem` in Cocoa Touch:

```
- initWithBarButtonSystemItem:target:action:
- initWithCustomView:
- initWithImage:style:target:action:
- initWithTitle:style:target:action:
- initWithImage:landscapeImagePhone:style:target:action:
```

Initializers may set various properties of the object being created; they also may take various routes to complete the initialization. If you create a series of initializers, there are two rules to follow:

- The first step in an initializer is to call [super `init`]. This ensures that the basic object is there and complete.
- Initializers can call other initializers that each performs some initialization. The most complete initializer (that is, the one that sets all of the properties rather than just some of them) is the designated initializer. The others are secondary initializers.

It is a good practice to put all of the initializers together in your code, possibly using a `#pragma mark –` section. This separates the initializers and, in the list of methods in the jump bar for a file, the pragma section names will appear.

Summary

In this hour, you have seen the messaging structure that is at the heart of Objective-C. It is not just another way of talking about the calling of methods that other languages use; it is a different way of constructing software. You see in later hours how the messaging architecture helps to implement dynamic aspects of Objective-C programs.

The common `alloc` and `init` methods have been used in this hour as examples. You have also seen how you can work with them and construct a hierarchy of designated and secondary initializers to construct your runtime objects.

Q&A

Q. **Why does `init` return an `id`?**

A. By returning an `id`, which is often a reference to the object on which it was called, it allows `init` to substitute a new object for the original one. Increasingly, the declarations are being changed to use the new `instanceType` return value, which allows for more error checking. Each new release of iOS adds more of these revisions.

Q. **What is the difference between a class method and an instance method?**

A. You must have an instance of a class to use an instance method; you can simply call a class method with the class name (that is, you do not need an instance). Class methods are often used as factory methods to create new instances of the class.

Workshop

Quiz

1. How do you differentiate class and instance methods in your code?
2. When do you call [super `init`] in an initializer?

Quiz Answers

1. Class methods begin with +; instance methods begin with –.
2. This is the first thing you do. You might go up through a hierarchy, but you have to have all of the superclasses created and initialized before you continue with your own.

Activities

Browse the Cocoa frameworks to see how messages are structured. Get used to the sequence of labeled parameters in the messages. It might take a while to be comfortable reading them, but remember that this is the environment in which you are going to be working.

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