Jesse Feiler

Sams Teach Yourself

Core Data for Mac® and iOS

in 24 Hours

SAMS

Jesse Feiler

Sams **Teach Yourself**

Core Data for Mac® and iOS

in 24 Hours

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

Sams Teach Yourself Core Data for Mac™ and iOS in 24 Hours

ISBN-13: 978-0-672-33577-8
ISBN-10: 0-672-33577-8

Library of Congress Cataloging-in-Publication data is on file.

Printed in the United States of America

First Printing: November 2011

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Loretta Yates

Development Editor
Sondra Scott

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Megan Wade

Indexer
Brad Herriman

Proofreader
Water Crest Publishing, Inc.

Technical Editor
Robert McGovern

Publishing Coordinator
Cindy Teeters

Designer
Gary Adair

Compositor
Mark Shirar

Contents at a Glance

Part V: Managing Data and Interfaces

Appendix

Table of Contents

About the Author

Jesse Feiler is a developer, web designer, trainer, and author. He has been an Apple developer since 1985 and has worked with mobile devices starting with Apple's Newton and continuing with the iOS products such as the iPhone, iPod touch, and iPad. Feiler's database expertise includes mainframe databases such as DMS II (on Burroughs), DB2 (on IBM), and Oracle (on various platforms), as well as personal computer databases from dBase to the first versions of FileMaker. His database clients have included Federal Reserve Bank of New York; Young & Rubicam (advertising); and many small and nonprofit organizations, primarily in publishing, production, and management.

Feiler's books include the following:

- *Sams Teach Yourself Objective-C in 24 Hours* (Sams/Pearson)
- *Data-Driven iOS Apps for iPad and iPhone with FileMaker Pro, Bento by FileMaker, and FileMaker Go* (Sams/Pearson)
- *Using FileMaker Bento* (Sams/Pearson)
- *iWork for Dummies* (Wiley)
- *Sams Teach Yourself Drupal in 24 Hours* (Sams/Pearson)
- *Get Rich with Apps! Your Guide to Reaching More Customers and Making Money NOW* (McGraw-Hill)
- *Database-Driven Web Sites* (Harcourt)
- *How to Do Everything with Web 2.0 Mashups* (McGraw-Hill)
- *The Bento Book* (Sams/Pearson)
- *FileMaker Pro In Depth* (Sams/Pearson)

He is the author of MinutesMachine, the meeting management software for iPad—get more details at champlainarts.com.

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

Acknowledgments

Thanks go most of all to the people at Apple, along with the developers and users who have helped to build the platform and imagine possibilities together to make the world better.

At Pearson, Loretta Yates, Acquisitions Editor, has taken a concept and moved it from an idea through the adventures along the way to printed books and eBooks in a variety of formats. She is always a pleasure to work with.

Mandie Frank, Project Editor, has done a terrific job of keeping things on track with a complex book full of code snippets, figures, and cross references in addition to the text. Technical Editor Robert McGovern caught numerous technical typos and added comments and perspectives that have clarified and enhanced the book.

As always, Carole Jelen at Waterside Productions has provided help and guidance in bringing this book to fruition.

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.

As an Editor-in-Chief for Sams Publishing, I welcome your comments. You can email or write me directly to let me 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 I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

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

Email: feedback@amspublishing.com

Mail: Greg Wiegand
Editor-in-Chief
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

This page intentionally left blank

Introduction

Organizing things is an important human activity. Whether it is a child organizing toys in some way (by size, color, favorites, and so forth) or an adult piecing together a thousand-piece jigsaw puzzle, the desire to "make order out of chaos" (as one inveterate puzzler put it) reflects a sense that somehow if we try hard enough or just have enough information, we can find or create an understandable view of the world. Or at least an understandable view of the left overs in the refrigerator or the photos in an album.

Core Data is a powerful tool that you can use with the Cocoa and Cocoa Touch frameworks on iOS and Mac OS to help you make order out of the chaos of the hundreds, thousands, and even billions of data elements that you now can store on your computer or mobile device.

Who Should Read This Book

This book is geared toward developers who need to understand Core Data and its capabilities. It's also aimed at developers who aren't certain they need the combination of Core Data and Cocoa. It places the technologies in perspective so that you can see where you and your project fit in. Part of that is simply analytical, but for everyone, the hands-on examples provide background as well as the beginnings of applications (apps) that you can create with these two technologies.

If you are new to databases or SQL, you will find a basic introduction here. If you are familiar with them, you will find a refresher as well as details on how the concepts you know already map to Core Data terminology.

Likewise, if you are new to development on Mac OS, iOS, or Cocoa and Cocoa Touch, you will find a fairly detailed introduction. If you are already familiar with them, you will see how some of the basic concepts have been expanded and rearranged to work with Core Data.

There is a theme that recurs in this book: links and connections between interface and code as well the connections between your app and the database. Much of what you find in this book helps you develop the separate components (interface, database, and code) and find simple ways to link them.

Some Points to Keep in Mind

Not everyone starts from the same place in learning about Core Data (or, indeed, any technology). Learning and developing with new technologies is rarely a linear process. It is important to remember that you are not the first person to try to learn these fairly complex interlocking technologies. This book and the code that you experiment with try to lead you toward the moment when it all clicks together. If you do not understand something the first time through, give it a rest, and come back to it another time. For some people, alternating between the graphical design of the interface, the logical design of the code processes, and the organization structure of the database can actually make things seem to move faster.

Here are some additional points to consider.

Acronyms

In many books, it is a convention to provide the full name of an acronym on its first use—for example, HyperText Markup Language (HTML). It is time to recognize that with wikipedia.org, dictionaries built into ebooks and computers, and so many other tools, it is now safe to bring a number of acronyms in from the cold and use them without elaboration. Acronyms specific to the topic of this book are, indeed, explained on their first use in any chapter.

There is one term that does merit its own little section. In this book, as in much usage today, SQL is treated as a name and not as an acronym. If you look it up on Wikipedia, you will see the evolution of the term and its pronunciation.

Development Platforms

It is not surprising that the development of Mac OS X apps takes place on the Mac itself. What may surprise some people, though, is that iOS apps that can run on iPad, iPod touch, and iPhone must be developed on the Mac. There are many reasons for this, not the least of which is that the development tool, Xcode, takes advantage of many dynamic features of Objective-C that are not available on other platforms. Also, Xcode has always served as a test bed for new ideas about development, coding, and interfaces for the Apple engineers. Registered Apple developers have access to preview versions of the developer tools. As a result, the Apple developers had access to features of Lion such as full-screen apps nine months before the general public. In fact, Xcode 4 is optimized for Lion in both speed and interface design.

Assumptions

Certain things are assumed in this book. (You might want to refer to this section as you read.) They are as follows:

- *Cocoa*, as used in this book, refers to the Cocoa framework on Mac OS and, unless otherwise specified, also to the Cocoa Touch framework on iOS.
- *iPhone* refers to iPhone and iPod touch unless otherwise noted.

Formatting

In addition to the text of this book, you will find code samples illustrating various points. When a word is used in a sentence as computer code (such as `NSTableView`), it appears like this. Code snippets appear set off from the surrounding text. Sometimes they appear as a few lines of code; longer excerpts are identified with listing numbers so they can be cross-referenced.

Downloading the Sample Files

Sample files can be downloaded from the author's website at northcountryconsulting.com or from the publisher's site at www.informit.com/9780672335778.

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 the book straight through. Like all of the *Teach Yourself* Books, as much as possible, each chapter (or 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 Core Data"

This part introduces the basic issues of the book and shows you principles and techniques that apply to all of the products discussed:

- Chapter 1, "Introducing Xcode 4"—Xcode is the tool you use to build Mac OS and iOS apps. It includes graphical editors for designing your interface and data model. The current version, Xcode 4, represents a significant step forward from previous development environments. You'll get started by learning the ins and outs of Xcode 4. After you use it, you'll never look back.

- Chapter 2, "Creating a Simple App"—This hour walks you through the process of creating an app from one of the built-in Xcode templates. It's very little work for a basic app that runs.
- Chapter 3, "Understanding the Basic Code Structure"—This hour introduces design patterns used in Objective-C as well as some of the features (such as delegates and protocols) that distinguish it from other object-oriented programming languages.

Part II, "Using Core Data"

Here you will find the basics of Core Data and its development tools in Xcode:

- Chapter 4, "Getting the Big Core Data Picture"—Here you'll find an overview of Core Data and a high-level introduction to its main components.
- Chapter 5, "Working with Data Models"—Data models have been around since the beginning of databases (and, in fact, since long before, if you want to include data models such as the classifications of plants and animals). This hour lets you learn the language of Core Data.
- Chapter 6, "Working with the Data Model Editor"—In this hour, you will learn how to build your data model graphically with Xcode's table and grid views.
- Chapter 7, "What Managed Objects Can Do"—In this hour, you'll discover the functionality of managed objects and what you can do to take advantage of it and to expand it.
- Chapter 8, "Controllers: Integrating the Data Model with Your Code"—The key point of this book is to show you how to link your database and data model to interface elements and your code. This hour provides the basics for Mac OS and for Cocoa.
- Chapter 9, "Fetching Data"—Just as the SQL `SELECT` statement is the heart of data retrieval for SQL databases, fetching data is the heart of data retrieval for Core Data. Here you'll learn the techniques and terminology.
- Chapter 10, "Working with Predicates and Sorting"—When you fetch data, you often need to specify exactly what data is to be fetched—that is the role of predicates. In addition, you will see how to build sorting into your fetch requests so that the data is already in the order you need.

Part III, "Developing the Core Data Interface"

Now that you understand the basics of Core Data, you can use it to drive the commands, controls, and interfaces of your apps:

- Chapter 11, "Finding Your Way Around Interface Builder: The Graphics Story"—The Interface Builder editor in Xcode 4 (a separate program until now) provides powerful tools and a compact workspace to help you develop your interface and app functionality.
- Chapter 12, "Finding Your Way Around Interface Builder: The Code Story"—This hour shows you the graphical tools to link the code to the interface.
- Chapter 13, "Control-Dragging Your Way to Code"—A special aspect of linking your interface to your code is using the tools in Xcode 4 to actually write the interface code for you.
- Chapter 14, "Working with Storyboards"—One of the major advances in Xcode 4, storyboards let you not only create and manage the views and controllers that make up your interface but also let you manage the sequences in which they are presented (`segues`). You will find that storyboards can replace a good deal of code that you would otherwise have to write for each view you display.

Part IV, "Building the Core Data Code"

Yet another aspect of the connections between Core Data, your code, and your interface consists of the data source protocol and table views. This hour explains them:

- Chapter 15, "Saving Data with a Navigation Interface"—Originally designed for iPhone, navigation interfaces are an efficient use of screen space for organized data. This hour shows you how to use them.
- Chapter 16, "Using Split Views on iPad"—Split views on iPad provide a larger-screen approach to data presentation than navigation interfaces. As you see in this hour, you can combine navigation interfaces with a split view on iPad. Data sources provide your Core Data data to the table view. This hour shows how that happens and moves on to how you can work with tables and their rows and sections. You'll also see how to format cells in various ways.

- Chapter 17, "Structuring Apps for Core Data, Documents, and Shoeboxes"—This hour goes into detail about how and where your data can actually be stored.
- Chapter 18, "Validating Data"—When you use Xcode and Core Data to specify what data is valid, you do not have to perform the validation yourself. This hour shows you how to set up the rules

Part V, "Managing Data and Interfaces"

- Chapter 19, "Using UITableView on iOS"—Table views let you manage and present data easily. The `UITableView` structure on iOS is designed for seamless integration with Core Data.
- Chapter 20, "Using NSTableView on Mac OS"—`NSTableView` on Mac OS is revised in Lion. The older versions of table views still work, but as you see in this hour, some of the new features of `UITableView` have been backported to Mac OS.
- Chapter 21, "Rearranging Table Rows on iOS"—The ability to rearrange table rows by dragging them on the screen is one of the best features of iOS. It is remarkably simple once you know the table view basics.
- Chapter 22, "Managing Validation"—This hour shows you how to build on the validation rules from Hour 18 to actually implement them and let users know when there are problems.
- Chapter 23, "Interacting with Users"—On both iOS and Mac OS, it is important to let users know when they are able to modify data and when it is only being displayed.
- Chapter 24, "Migrating Data Models"—You can have Core Data automatically migrate your data model to a new version. This hour shows you how to do that, as well as how to use model metadata and alternative types of data stores.

Appendixes

- Appendix A, "What's Old in Core Data"—There are some legacy features in the sample code you'll find on developer.apple.com and in apps you might be working with. This appendix helps you understand what you're looking at and how to modernize it.

HOUR 3

Understanding the Basic Code Structure

What You'll Learn in This Hour:

- Exploring the world of Objective-C
- Getting inside Objective-C objects
- Managing inheritance
- Using delegates and protocols
- Using model/view/controller

Working with the Code

Mac OS and iOS apps are written using the Objective-C language. Right there, some people might panic and throw up their hands, but do not worry. As pointed out previously, you write very little code from scratch. Much of the code that you run is already written for you using Objective-C; that code is in the Cocoa and Cocoa Touch frameworks that support everything from animation to native platform appearance and the Core Data and various table view classes that are the topic of this book. (Cocoa Touch is the version that runs on iOS; unless otherwise noted, references to Cocoa include Cocoa Touch in this book, just as references to iPhone include iPod touch.)

When you're working inside the Cocoa framework and the other components of iOS and Mac OS, most of your work consists of calling existing methods and occasionally overriding them for your own purposes. Xcode 4 provides a new development environment that is heavily graphical in nature. You will find yourself drawing relationships in your data model and, in the interface, drawing connections between objects on the interface and the code that supports them.

NOTE

Actually, Cocoa is an ever-evolving set of frameworks. You can find an overview at http://developer.apple.com/technologies/mac/cocoa.html.

Blank pages are rarely part of your development environment.

NOTE

This hour provides an overview of Objective-C. It provides some comparisons to other object-oriented languages such as C++, but its focus is on Objective-C and, particularly, in the ways in which it differs from object-oriented languages you might already know. You can find many introductions to object-oriented programming on the Web and in bookstores, so if you are unfamiliar with that basic concept, you might want to get up to speed on the basics.

Objective-C 2.0

First announced at the 2006 Worldwide Developers Conference and released in Mac OS X v.10.5 (Leopard) in October 2007, Objective-C 2.0 is now the standard implementation. It is fully supported in Xcode 4. The primary changes from the original version of Objective-C include the addition of automatic garbage collection, improvements to runtime performance, and some syntax enhancements. Those syntax enhancements are fully reflected in this book (after all, this book is written more than five years after the announcement of Objective-C 2.0). Some legacy software still uses old syntax, and there is generally no problem with that.

Objective-C 2.0 is often referred to as *modern,* while the previous version is referred to as *legacy*. The modern version is not to be confused with Objective-C *modern syntax*, a project in the late 1990s that changed the presentation of its syntax and which was ultimately discontinued.

As a general rule, legacy Objective-C code runs without changes in the Objective-C environment (there are some exceptions). Much of the sample code for Mac OS X on developer.apple.com is from the legacy period and, with few exceptions, it compiles and runs well. During the transition period, developers often continued to use legacy syntax. This meant that for shared code (and for sample code), developers did not have to worry about whether the code would be compiled or run in the modern or legacy system—it would generally work.

Today, there is no reason to write legacy code because the tools are all updated to Objective-C 2.0. It is safe to write code that will not compile or run in the legacy environment because people are not (or should not be) still using it.

- This is particularly relevant to *declared properties,* which are discussed later in this hour in the "Using Declared Properties" section, p. 68.

What You Do Not Have to Worry About

You do not have to worry about designing an entire program in most cases. You are writing code that will be a part built on a template, the behavior of which is known by users, so what you have to do is to fit in. You need to write the code that is specific to your app, but you do not have to worry about implementing an event loop.

In fact, if you decide to develop the app's infrastructure yourself, you might find that users are disappointed at its unfamiliarity and—more important to many people—your app might not find a place in the App Store.

Instead of writing code from scratch, much of what you will do is to investigate the code that you have in the Xcode templates or in Apple's sample code. You need to explore what is written and how it has been designed so that you can understand how and where your functionality will fit in. It is a very different process than writing it from scratch.

Introducing Objective-C

Objective-C is built on C; in fact, if you write ordinary C code, you can compile it with an Objective-C compiler (that includes Xcode). The main extension to C that Objective-C provides is its implementation of objects and object-oriented programming.

Today, object-oriented programming rates a big yawn from many people; that is the kind of programming that most people are used to. When Simula 67, the first precursor of Objective-C and all modern object-oriented languages, was developed, this was a new notion, and many people were not certain it was worth the extra effort (not to mention the time it took to learn what then was a new and not fully formed technology). It is against this background that the extensions to C needed to implement Objective-C were created. One of the goals was to prove that very little was needed to be added to C to implement object-oriented programming.

Basically, what was added to C was a messaging and object structure based on Smalltalk. Over the years, additional features such as protocols and delegates as well as categories and blocks were added to the language. Some other features were added. Some of them are not as important to developers writing for iOS or Mac OS X, while others of them simply never caught on with developers at large. Thus, this section provides an overview of the major components that are in use today in the context of iOS and Mac OS X.

At the same time as additional features were being added, the use of the language was refined particularly in the environments of NeXT, Apple, Mac OS X, and iOS.

These refinements include conventions such as naming conventions and even code formatting conventions. They are not part of the language itself, but they represent best practices that are followed by the vast majority of Objective-C developers.

Looking at Object-Oriented Programming in the Context of Objective-C

The heart of the implementation of object-oriented programming consists of the objects themselves and the roles that they can play. With Objective-C, there is another point to notice about the implementation of the language; because it is a *dynamic* language, some of the work that would be done in the compile and build process for a language such as C++ is done at runtime. This means that the runtime environment, which, for all intents and purposes is the operating system, is a much bigger player than it is in other languages.

Differentiating Classes, Instances, and Objects

The first point to remember is that objects, classes, and instances are related but different concepts. These concepts exist in most object-oriented languages:

- **Class**—A *class* is what you write in your code. It typically consists of a header file (ending in .h) with an *interface,* as well as an implementation file (ending in .m) that provides the code to support the interface.
- **Instance**—At runtime, a class can be *instantiated*. That converts it from instructions in your program to an object that has a location in memory and that can function.
- **Object**—*Object* is a term that is commonly used in contexts where most people understand what is meant. The word can be used to refer to instances or classes, but most of the time, it refers to instances.

Understanding What Is Not an Object

Some basic types are declared in Foundation/Foundation.h. Each of these is implemented as a struct (`NSDecimal`), a typedef (`NSUinteger`), or an enum (`NSComparisonResult`). Sometimes these hide the actual implementation, such

as this definition of NSInteger, which resolves to a long on a 64-bit application and to an int otherwise:

```
#if __LP64__ || TARGET_OS_EMBEDDED || TARGET_OS_IPHONE ||
  TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
#else
typedef int NSInteger;
#endif
```

Using these types makes your code more maintainable and portable than using native C types.

NOTE

The NS prefix refers to NeXTSTEP.

With the exception of basic types such as these, almost everything you deal with is an object. You will find some non-object entities in the Core Foundation framework and in specialized frameworks that often deal with low-level operations such as Core Animation.

Understanding the Three Purposes of Objects

Building on Smalltalk's structure, objects in Objective-C have three purposes and functions:

- **State**—Objects can contain *state,* which in practical terms means that they can contain data and references to other objects. In implementation and use, state usually consists of member variables, instance data, or whatever terminology you use.
- **Receive messages**—Objective-C objects can receive messages sent from other objects.
- **Send messages**—Objective-C can send messages to other objects.

- Communication between objects is via these messages, which are highly structured. This is covered in more detail in the "Messaging in Objective-C" section later in this hour, p. 73.

Data is *encapsulated* within objects in Objective-C. That means it is accessible only through messages. Objects cannot access another object's data (state) directly as can happen in other object-oriented languages. This is a general goal of all good object-

oriented programming, but it is enforced in Objective-C in ways that are often best practices in other languages.

Declared properties, which are discussed in the next section, shows you how this is done in Objective C. This section also explains why, at first glance, it can appear that you can access internal data from another object and why, at second glance, you will see that it is only the appearance of direct access.

Using Declared Properties

One of the most significant new features of Objective-C 2.0 was the introduction of *declared properties*. The concept is quite simple and eliminates a great deal of tedious typing for developers. The feature is best demonstrated by showing before-and-after examples.

Declaring a Property

Listing 3.1 shows a typical interface using legacy syntax. This is the .h file, and it contains the interface in a section starting with the compiler directive `@interface`. `@` always introduces compiler directives; note the `@end` at the end of the file. Interface code can appear in other places, but the .h file for a class is the primary place.

NOTE

Notice that because the code in Listing 3.1 was generated by Xcode, the comments are automatically inserted.

LISTING 3.1 Legacy Class Declaration

```
//
//  My_First_ProjectAppDelegate.h
//  My First Project
//
//  Created by Sams on 6/14/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.
//

#import <Cocoa/Cocoa.h>

@interface My_First_ProjectAppDelegate : NSObject <NSApplicationDelegate> {
@private
  NSWindow *window;
}

 - (IBAction)saveAction:sender;
@end
```

The class shown here, `My_First_ProjectAppDelegate`, has an interface with one variable. After that, a single method, `(IBAction)saveAction:sender`, is declared. By convention, variable names that begin with underscores are private and should not be used directly. Also, note that all the variables are references—the `*` indicates that at runtime, a reference to the underlying object's structure is to be used and resolved as needed. The `@private` directive means that these variables are private to this class; by contrast, `@protected` would allow descendants of this class to use them. `@public`, which is rarely used (and which is considered poor syntax), allows any object to access these variables directly. This syntax is described later in this hour.

As you can see, there is no method declared that will allow another object to access the data inside this object. Listing 3.2 adds *accessor* methods to access the data. These are referred to generally as *accessors* and specifically as *getters* or *setters*. By using this coding best practice, the variables are encapsulated and can be accessed only through these methods.

LISTING 3.2 Legacy Class Declaration with Accessors

```
//
//  My_First_ProjectAppDelegate.h
//  My First Project
//
//  Created by Sams on 6/14/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.
//

#import <Cocoa/Cocoa.h>

@interface My_First_ProjectAppDelegate : NSObject <NSApplicationDelegate> {
@private
  NSWindow *window;
}

- (NSWindow*) getWindow;
- (NSWindow*) setWindow: (NSWindow*)newindow;

- (IBAction)saveAction:sender;
@end
```

Listing 3.3 demonstrates the use of declared properties in Objective C 2.0.

LISTING 3.3 Modern Class Declaration

```
//
//  My_First_ProjectAppDelegate.h
//  My First Project
//
//  Created by Sams on 6/14/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.
//
```

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

@interface My_First_ProjectAppDelegate : NSObject <NSApplicationDelegate> {
}

@property NSWindow* window;

- (IBAction)saveAction:sender;
@end
```

The individual declarations of the variables are gone; they are replaced by *declared properties* that are implemented with compiler directives. As compiler directives, these are merely instructions to the compiler. They are not part of the program's syntax.

Synthesizing a Property

A declared property directive works together with a companion synthesize directive that appears in the implementation file. The companion synthesize directive to the property declaration is shown in Listing 3.4.

LISTING 3.4 Synthesize Directives to Match Listing 3.3

```
@synthesize window;
```

When the program is compiled, these two directives generate code. If, as in Listing 3.3, the variables are not declared, the declarations are created. They will look just like the code that has been typed into Listings 3.1 and 3.2. In addition, getters and setters will be automatically generated. They will look exactly like those typed at the bottom of Listing 3.2.

And, perhaps most important, the declared properties allow for the use of *dot syntax* that automatically invokes the relevant accessors. It also provides the appearance of direct access to the encapsulated data of the object. Given the code in Listings 3.3 and 3.4, you could write the following code to reference the data within an object of type `My_First_ProjectAppDelegate` that has been instantiated with the name `jf_My_First_ProjectAppDelegate`:

```
jf_ My_First_ProjectAppDelegate.managedObjectContext
```

The appropriate accessor (getter or setter) will be invoked as needed. Note that within the implementation code of an object, you can always use `self` to refer to the object itself. Thus, you can write

```
self.managedObjectContext = <another managedObjectContext>;
```

Or

```
<myManagedObjectContext > = self.managedObjectContext;
```

You save a great deal of typing and make your code much more readable by using declared properties.

You can still declare the variables if you want to. At compile time, the same-named variables you have declared will be accessed by the property. However, a common use of properties is to reinforce the hiding of internal variables. The property declaration can provide a name that is used by programmers while the underlying variable is not accessed. This is common in the framework code you deal with.

For example, here is a declaration of a private variable:

```
NSWindow *__window;
```

Here is a companion property declaration:

```
@property nonatomic, retain, readonly NSWindow* window;
```

The synthesize directive would normally create the `window` variable because there is none. However, you can use the following synthesize directive to have the property's accessors, which are created during compilation by the synthesize directive, point to `__window` if you have declared it, as shown in Listing 3.5.

LISTING 3.5 Using a Private Variable in a Property

```
@synthesize window = __window;
```

You can access the private variable by using its name if you are allowed to do so, which in practice generally means for code in the class itself. Thus, you can write:

```
__window = <something>;
```

Using dot syntax, you go through the property and, as a result, the following code can have the same effect:

```
self.window = <something>;
```

NOTE

There is one case in which the direct access with dot syntax does have a difference. If you have set a variable to an object that has been allocated in memory, the appropriate way to set it to another value is to dispose of the first object and then set it again. Disposing of an object that is no longer needed prevents *memory leaks*—the bane of developers. Because the accessors can perform any operations you want, they can dispose of no-longer-needed objects as part of their setting process.

In practice, synthesize directives usually are a bit more complex. You can provide *attributes* by placing them in parentheses after the property directive. A common set of attributes in a synthesize directive is the following:

```
property (nonatomic, strong, readonly)
  NSPersistentStoreCoordinator *persistentStoreCoordinator;
```

Attributes reflect the reality of today's environment and the features of modern Objective-C. It is no longer enough to know that a variable is of a specific type. Many other attributes come into play, and the property directive allows you to set them. Its syntax also allows for the expansion of attributes in the future as the language evolves. Thus, property directives together with the appropriate attributes combine to create rich, useful objects that are easy to use and maintain over the lifespan of the app.

Table 3.1 shows the current set of attributes and the available values. The default values (having a synthesize directive create the accessors, assign, and atomic) are most commonly used. Notice also that the opposite of the atomic attribute is to omit it—in other words, there is no separate "nonatomic" attribute. (Over time, these attributes have changed to add new features. Consult the release notes for new versions of Xcode for these changes.)

TABLE 3.1 Attributes for Declared Properties

Attribute	Values	Notes
Accessor	`getter = <name of your getter>`	Accessors are synthesized for you unless you provide your own.
	`setter = <name of your setter>`	You can go further by specifying your own custom accessors. You will have to write them, but it might be worthwhile in special cases.
Writability	`readwrite`	
	`readonly`	
Setter	`assign` (default)	
Semantics	`retain`	Retains the object after assignment and releases the previous value.
	`copy`	Copies the object and releases the previous value.
Atomicity	`nonatomic`	Default is `atomic` so that getters and setters are thread-safe.

Using Dynamic Properties

Instead of a synthesize directive, you can use a dynamic directive for any property. The format is as follows:

```
@dynamic myValue;
```

The dynamic directive indicates that your code is going to be providing the appropriate values for the property at runtime. This entails writing some rather complex code, but there is an alternative. Core Data implements the functionality promised by the dynamic directive, so you do not have to worry—just keep reading.

Messaging in Objective-C

Objective-C is a *messaging* environment, not a *calling* environment. Although the end result is very much the same as in a calling environment, you send a message to an object in Objective-C. That message consists of an object name and a method of that object that can respond to the message. Here is an example:

```
[myGraphicsObject draw];
```

In a language such as C++, you would call a method of the object, as in:

```
myGraphicsObject.draw();
```

NOTE

There is a lot of information in Apple's documentation as well as across the Web detailing the technical differences and how the two styles evolved. The most important point to remember is that a primary purpose of Objective-C was to show that object-oriented programming could be implemented very simply with a small set of Smalltalk-based variations on top of C. More than three decades later, whether Objective-C is simpler than C++ is a topic of much debate (although many reasonable people have moved on to other matters).

The arguments sent to a function (or method) in C are placed in parentheses, and their sequence is determined by the code. In Objective-C, the arguments are named. Thus, a C-style function looks like this:

```
resizeRect (float height, float width){
  return height * width;
}
```

If that function were part of an object in C++, you would invoke it with the following:

```
myRect.resizeRect (myHeight, myWidth);
```

An Objective-C-style function looks like this:

```
-(void) resizeRect: (float*) height newWidth:(float*)width {
}
```

You invoke it with the following:

```
[myRect resizeRect: myHeight newWidth: myWidth];
```

The most important difference you notice is that the parameters in Objective-C are labeled, whereas the parameters in C or C++ are strictly positional. Do not be misled: The labels do not imply that the order of the parameters can vary. The following Objective-C function is not identical to the previous one because the labels (that is, the order of the parameters) are different:

```
-(void) resizeRect: (float*)width newHeight:(float*)height {
```

WARNING

Technically, the labels before the colons are optional. Omitting them is a very bad practice. It is best to assume that they are required in all cases.

Despite the fact that there are many similarities, you will find it easier to learn Objective-C if you avoid translating back and forth to and from other programming languages you know. The principle is just the same as it is for learning a natural language—start thinking in the new language right away.

Naming Conventions

The definitive reference about naming items in your code is in "Coding Guidelines for Cocoa" located at http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html. These are detailed guidelines to help you make your code consistent with best practices and standards.

There is one guideline that is sometimes an issue. As noted previously in this hour, you can begin an instance variable name with an underscore to indicate that it is private. In some documentation, developers are advised that only Apple is to use this naming convention. That is all well and good, but some of the Xcode templates and example code do use underscores at the beginning of names for private variables. As you expand and modify your projects built on these templates and examples, you may choose to make your code consistent so that the naming convention is the same for the variables named by Apple in the template and those named by you that are syntactically parallel. Just be aware of the issue, and make your choice.

The other naming conventions are normally adopted by most developers. When there are deviations, it often is the case that a developer is not aware of the conventions.

Using Protocols and Delegates

Many people consider protocols and delegates to be advanced topics in Objective-C. However, as you will see, they are critical in the table views that are often used to manipulate Core Data objects as well as in Core Data itself. They are also used throughout the iOS and Mac OS frameworks. This section explains that there are several pieces to the puzzle, but they fit together the same way in every case. Once you've worked through a few of them, they will become very natural. In particular, you will see that a lot of the details need no attention from you when you use a protocol or delegate. This section shows you how they work, but soon you will appreciate the fact that they are another part of the operating system that just works without too much of your attention.

Looking Up the Background of Protocols and Delegates

Object-oriented programming offered (and continues to offer) very powerful ways to build and maintain code. One issue arose quite early—multiple inheritance. For simple classroom examples, it is easy to propose a base class of Toy, with subclasses of Ball, Jump rope, and Puzzle. It is also easy to propose a base class of Sports Equipment with subclasses of Ball, Jump rope, Puzzle, Net, and Score board.

Both of these object hierarchies refer to real-life objects, and both make sense to most people. However, as soon as you start programming with those objects, you might find that you want an object such as a ball to have some variables and behaviors that descend from Toy and some that descend from Sports Equipment. In other words, can a ball have two superclasses (or ancestors)?

Many proposals have been made and implemented for solving the multiple inheritance problem. Objective-C started out addressing that issue and has evolved a structure that handles multiple inheritance. However, it also covers a number of other long-time object-oriented programming issues.

NOTE

This overview of history is not only simplified, but also benefits from 20/20 hindsight so that it is now possible to construct plausible and rational sequences of events. At the time that they occurred, though, the overall picture was not yet visible to the participants.

The Objective-C approach that has evolved allows you to share functionality between two objects without using inheritance. To be sure, inheritance is used throughout Objective-C, but the very deep inheritance hierarchies that often evolve in languages such as C++ are far less common in Objective-C. Instead, you can take a defined chunk of functionality and share it directly.

A major distinction between extending a class by subclassing it and extending a class by adding a protocol to it is that a subclass can add or modify methods and can also add new instance variables. Protocols, like categories that are described briefly at the end of this section, only add methods.

Using an Example of a Protocol

An example of this is found in the iOS sample code for Multiple Detail Views (http://developer.apple.com/library/ios/#samplecode/MultipleDetailViews/). This code addresses an issue that arises with some iPad apps. iOS supports a split view in which the main view fills the screen when the device is vertically oriented; when the device is horizontally oriented, the right and larger part of the screen shows the detail view, but, at the left, a list of items controls what is shown in the larger view.

In the vertical orientation, a control bar at the top of the window contains a button that will let you open a popover with the list of items that can be shown at the left.

The problem arises because the control bar at the top of the window can be a navigation bar or a toolbar. These are two different types of controls. The button to bring up the popover needs to be shown (in portrait mode) and hidden (in landscape mode). The code to implement this differs whether the button is added to a toolbar or to a navigation bar.

The key to this consists of four steps:

- **Declaring a protocol**—A protocol is declared. It is a set of methods presented as they would be in an interface.
- **Adopting the protocol**—Any class in this sample app that wants to be able to use this protocol must *adopt* it in its header. Adopting a protocol means that the class declared in the header must implement methods from the protocol. (Note that ones marked optional do not have to be implemented. This is another Objective-C 2.0 improvement.)
- **Implementing the protocol**—Any class that adopts the protocol must implement all required methods and might implement other methods. The implementations might use variables and other methods of the particular class that adopts the protocol.
- **Using the protocol.**

The code is described in the following sections.

The first step is to define the protocol in RootViewController.h, as shown in Listing 3.6.

LISTING 3.6 Defining the Protocol

```
@protocol SubstitutableDetailViewController
- (void)showRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem;
- (void)invalidateRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem;
@end
```

Beginning with Objective-C 2.0, you can indicate which methods are required or optional. The default is required, so the code in Listing 3.6 actually is the same as the code shown in Listing 3.7.

LISTING 3.7 Marking Protocol Methods Required or Optional

```
@protocol SubstitutableDetailViewController
@required
- (void)showRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem;
- (void)invalidateRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem;
@end
```

After you have declared a protocol, you need to adopt it. Listing 3.8 shows the code from the sample app for a view with a toolbar. Although the protocol is declared in RootViewController.h, it is adopted in FirstDetailViewController.h (and in the second one, too).

LISTING 3.8 Protocol Adoption with a Toolbar

```
@interface FirstDetailViewController : UIViewController <
SubstitutableDetailViewController> {

UIToolbar *toolbar;
}
```

Listing 3.9 shows the protocol adopted by another view that uses a navigation bar.

LISTING 3.9 Protocol Adoption with a Navigation Bar

```
@interface SecondDetailViewController : UIViewController <
SubstitutableDetailViewController> {

UINavigationBar *navigationBar;
}
```

Each of the classes that has adopted the protocol must implement its methods. Listing 3.10 shows the implementation of the protocol with a toolbar in FirstDetailViewController.m.

LISTING 3.10 Implementation of the Protocol with a Toolbar

```
#pragma mark -
#pragma mark Managing the popover

- (void)showRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem {


// Add the popover button to the toolbar.

NSMutableArray *itemsArray = [toolbar.items mutableCopy];
  [itemsArray insertObject:barButtonItem atIndex:0];
  [toolbar setItems:itemsArray animated:NO];
  [itemsArray release];
}

- (void)invalidateRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem {


// Remove the popover button from the toolbar.

NSMutableArray *itemsArray = [toolbar.items mutableCopy];
  [itemsArray removeObject:barButtonItem];
  [toolbar setItems:itemsArray animated:NO];
  [itemsArray release];
}
```

In Listing 3.11, you see how you can implement the protocol with a navigation bar. (This code is from SecondDetailViewController.m.)

NOTE

In Listing 3.11, you will also see how certain types of operations, such as adjusting buttons on a navigation bar, can be easier than the corresponding operations on a toolbar.

LISTING 3.11 Implementation of the Protocol with a Navigation Bar

```
#pragma mark -
#pragma mark Managing the popover

- (void)showRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem {

// Add the popover button to the left navigation item.
  [navigationBar.topItem setLeftBarButtonItem:barButtonItem animated:NO];
}

- (void)invalidateRootPopoverButtonItem:(UIBarButtonItem *)barButtonItem {

// Remove the popover button.
  [navigationBar.topItem setLeftBarButtonItem:nil animated:NO];
}
```

The next step is to adopt another protocol. This protocol, `UISplitViewController Delegate`, is part of the Cocoa framework, so you do not have to write it. All you have to do is adopt it as the `RootViewController` class in the example does. The Interface is shown in Listing 3.12 together with the adoption of the protocol in RootViewController.h. To repeat, what that adoption statement (in the < and >) means is that all required methods of the protocol will be implemented by this class.

TIP

In addition, remember that without an `optional` directive, all methods are required.

LISTING 3.12 Adopting the `UISplitViewControllerDelegate` Protocol

```
@interface RootViewController : UITableViewController
  <UISplitViewControllerDelegate> {
UISplitViewController *splitViewController;


UIPopoverController *popoverController;
UIBarButtonItem *rootPopoverButtonItem;
}
```

Having promised to implement the required and (possibly) optional methods of the `UISplitViewControllerDelegate` protocol, `RootViewController.m` must do so. The sample app implements two of the methods as shown in Listing 3.13. In doing so, it has fulfilled the promise made when it adopted the `UISplitViewController-Delegate` protocol.

There are two critical lines, one in each method of Listing 3.13. Those lines are the same in both methods and are underlined. It is easiest to start reading them from the middle. The heart of each line is the assignment of a local variable, `*detailViewController`, using the split view controller's array of view controllers and selecting item one.

This local variable is declared as being of type `UIViewController` and adopting the `SubstitutableDetailViewController` protocol shown previously in Listing 3.6. Because it adopts the protocol, it is safe to assume that it implements all the required methods. Because nothing is marked optional, both methods are required, so it is certain that they will be there (if they are not, that assignment statement will fail).

LISTING 3.13 Implementing the protocol in `RootViewController.m`

```
- (void)splitViewController:(UISplitViewController*)svc
  willHideViewController:(UIViewController *)aViewController
  withBarButtonItem:(UIBarButtonItem*)barButtonItem
  forPopoverController:(UIPopoverController*)pc {

// Keep references to the popover controller and the popover button, and tell the
// detail view controller to show the button.
  barButtonItem.title = @"Root View Controller";

self.popoverController = pc;

self.rootPopoverButtonItem = barButtonItem;

UIViewController <SubstitutableDetailViewController> *detailViewController =
  [splitViewController.viewControllers objectAtIndex:1];
  [detailViewController showRootPopoverButtonItem:rootPopoverButtonItem];
}

- (void)splitViewController:(UISplitViewController*)svc
  willShowViewController:(UIViewController *)aViewController
  invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem {

// Nil out references to the popover controller and the popover button, and tell
// the detail view controller to hide the button.

UIViewController <SubstitutableDetailViewController> *detailViewController =
      [splitViewController.viewControllers objectAtIndex:1];
      [detailViewController invalidateRootPopoverButtonItem:rootPopover
        ButtonItem];

self.popoverController = nil;

self.rootPopoverButtonItem = nil;
}
```

You might have to trace through the code again, but it is worth it to get the hang of it. The point is that this locally declared class inherits from a standard class in the framework (`UIViewController`). However, by creating and adopting its own protocol, two separate classes with two different ways of implementing control bars can both promise to do the same thing, albeit in different ways because they have different types of control bars to work with.

NOTE

There is a related concept in Objective-C, called categories. A *category* consists of methods (no instance variables, just like protocols) that are added to a specific class. This allows people to modify a class without having access to the code that is used for it to perform its work. At runtime, there is no difference between the basic class and the methods that have been added with a category.

Using Delegates

Protocols are often paired with *delegates*, another key Objective-C concept. As noted previously in this hour, instead of calling procedures, messages are sent to objects in Objective-C. That makes the use of delegates possible. A class can declare a delegate for itself. That delegate processes messages sent to the object itself. Frequently, functionality is wrapped up in a protocol as you have seen here, and some of those protocols are designed to be used by delegates.

NOTE

One of the most common uses of a delegate is a delegate for the application class. Messages send to the application are passed along to the application delegate. This enables you to add functionality to an application without subclassing it: you just add your new functionality to the delegate.

For example, you saw in Listing 3.12 that the `RootViewController` class adopts the `UISplitViewControllerDelegate` protocol. This means that a `RootViewController` can be named as the delegate of an object that requires that protocol to be implemented.

- This is a high-level view of delegates and protocols. You will find more examples and much more detail in Part IV, "Using Data Sources and Table Views." If it is a little fuzzy now, do not worry.

Using the Model/View/Controller Concepts

One of the critical pieces of iOS and Mac OS is the model/view/controller (MVC) design pattern. Along with object-oriented programming, this is another concept that evolved in the heady days of the 1970s and 1980s when the technology world was addressing the rapidly changing environment in which personal computers were becoming more prevalent and vast numbers of people started using their own computers (and programming them).

Model/view/controller got a frosty reception from some people at the start because it seemed like an over-complicated academic exercise. In retrospect, it seems that perception might have arisen in some cases because the benefits of MVC only become apparent when you apply the pattern and concepts to large systems. Writing "Hello World" using MVC concepts is indeed over-complicated. However, writing Mac OS or iOS in Basic, COBOL, or even C would be something close to futile.

Fortunately, we have complex problems and powerful computers today and MVC has come into its own. The concepts are quite simple:

- **Model**—This is the data you are working with. With Core Data, you will always have a data model.
- **View**—This is the presentation of the model and the interface for users to manipulate it.
- **Controller**—This code is the glue between model and view. It knows details of both, so if either changes, the controller normally needs to be changed, too. However, a change to the view typically does not require a change to the model, and vice versa. The addition of new data to the model will require a change to the view, but that is only because the underlying reality affects both.

One of the mistakes people made early on was to draw a conceptual diagram with a large bubble for the model and another large bubble for the view. The controller was relegated to a small link. In practice, that is far from the case. The controller is often the largest set of code modules. For people used to traditional programming, working on a controller feels most like traditional programming. Both the model and the view can be highly structured, but it is in the controller that all the idiosyncrasies emerge and collide.

Importing and Using Declarations in Files

A compiler instruction that you have seen in many of the code snippets in this hour is `import`. It is broadly similar to the C `include` statement but bypasses some of the issues that arise with multiple uses of the `include` statement. In Objective-C, `import` checks to see if the file has already been imported and does not repeat the import if it is unnecessary. Files within your project are identified by their filenames enclosed in quotes; files that are part of the frameworks are enclosed in < and >.

Typically, an interface (.h) file imports other interface files. An implementation file (.m) imports it own interface file, and it might import other interface files (usually not implementation files).

In an interface file, it is common to declare protocols and classes that will be defined later in the build process. Thus, instead of using

```
#import "myclass.h"
```

you might be better off simply declaring

```
@class myclass
```

The templates and samples demonstrate this style in many cases.

Summary

This hour provides an introduction to Objective-C and its concepts, as well as a general comparison to other object-oriented languages you might know. The biggest differences from other object-oriented languages are its messaging syntax (rather than function calling syntax), the ability to extend code in ways other than subclassing (protocols, delegates, and categories), and its somewhat more rigorous enforcement of basic object-oriented design principles when compared to languages such as C++ (this last point remains a topic of much discussion and dissension).

You have seen some examples of Objective-C code at work, and you will see more—and write more—throughout this book. For some people, the syntax and nomenclature is daunting with all its square brackets. Have no fear; if you start to use it, you will soon become accustomed to it. Furthermore, that syntax helps you access some of the powerful features of Objective-C that have no direct parallels in other programming languages.

Q&A

Q. ***Where is the best place to start learning about model/view/controller?***

A. Start with controllers and, in particular, start with some of the view controllers such as the ones described in Part IV. The basics of both the view and model components of MVC are quite simple—one is your data and the other is your interface. The controller is pretty much where all the programming you are used to takes place.

Q. ***Are there naming conventions for methods and instances?***

A. Naming conventions for instance variables are usually dependent on the project or developer. Inside the frameworks, you will find a number of standard types of methods. These are typically implemented in subclasses of the major framework classes. They have names such as `viewWillAppear` and `viewDidAppear` so that you can insert your code at the right place. The documentation on developer.apple.com helps you to understand which parts of the creation of objects such as views (and the corresponding destruction) are done in which step.

Workshop

Quiz

1. *What is one of the biggest differences between subclassing and extending a class with a protocol or category?*
2. *What are the benefits of using declared properties?*

Quiz Answers

1. You can add instance variables to a subclass. Protocols and categories allow you to only add methods.
2. You can save yourself some typing, your code will be more readable, you can use dot syntax to use the accessors, and the attributes you can assign to the properties can help you take advantage of runtime features.

Activities

If you have not already done so, create a new Xcode project from one of the templates. Go through the code and pick out the Objective-C constructs that have been discussed in this hour. Be certain to run the code! Only hands-on experience with the code will help you understand what the code is doing.

Explore one or more of the sample code projects on developer.apple.com (these are often more complex than the Xcode projects). You might want to search for specific features, such as protocols or delegates, that you want to explore.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

S

T

U

V

W

X