Stephen G. Kochan

Updated for iOS5 and ARC

Programming in Objective-C

Fourth Edition

Developer's Library

Programming in Objective-C

Fourth 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: 978-0-672-32756-8

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

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

Programming in Objective-C

Fourth 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, Fourth Edition

ISBN-13: 978-0-321-81190-5

ISBN-10: 0-321-81190-9

Library of Congress Cataloging-in-Publication Data

Kochan, Stephen G.

Programming in objective-c / Stephen G. Kochan. – 4th ed.

p. cm.

ISBN 978-0-321-81190-5 (pbk.)

1. Objective-C (Computer program language) 2. Object-oriented programming (Computer science) 3. Macintosh (Computer)–Programming.

I. Title.

QA76.64.K655 2012

005.1'17–dc23

2011046245

Printed in the United States of America

First Printing December 2011

Trademarks

Warning and Disclaimer

Bulk Sales

Pearson 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

Acquisitions Editor
Mark Taber

Development Editor
Michael Thurston

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Indexer
Heather McNeill

Proofreader
Sheri Cain

Technical Editors
Wendy Mui
Michael Trent

Publishing Coordinator
Vanessa Evans

Designer
Gary Adair

Compositor
Mark Shirar

To Roy and Ve, two people whom I dearly miss.

To Ken Brown, "It's just a jump to the left."

Contents at a Glance

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 Reviewers

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 cocoadev.com website, 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 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.

Email: 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.

1

Introduction

Dennis Ritchie at AT&T Bell Laboratories pioneered the C programming language in the early 1970s. However, this programming language did not begin to gain widespread popularity and support until the late 1970s. This was because, until that time, C compilers were not readily available for commercial use outside of Bell Laboratories. Initially, this growth in popularity was also partly spurred by the equal, if not faster, growth in popularity of the UNIX operating system, which was written almost entirely in C.

Brad J. Cox designed the Objective-C language in the early 1980s. The language was based on a language called SmallTalk-80. Objective-C was *layered* on top of the C language, meaning that extensions were added to C to create a new programming language that enabled *objects* to be created and manipulated.

NeXT Software licensed the Objective-C language in 1988 and developed its libraries and a development environment called NEXTSTEP. In 1992, Objective-C support was added to the Free Software Foundation's GNU development environment. The copyrights for all Free Software Foundation (FSF) products are owned by the FSF. It is released under the GNU General Public License.

In 1994, NeXT Computer and Sun Microsystems released a standardized specification of the NEXTSTEP system, called OPENSTEP. The Free Software Foundation's implementation of OPENSTEP is called GNUStep. A Linux version, which also includes the Linux kernel and the GNUStep development environment, is called, appropriately enough, LinuxSTEP.

On December 20, 1996, Apple Computer announced that it was acquiring NeXT Software, and the NEXTSTEP/OPENSTEP environment became the basis for the next major release of Apple's operating system, OS X. Apple's version of this development environment was called Cocoa. With built-in support for the Objective-C language, coupled with development tools such as Project Builder (or its successor Xcode) and Interface Builder, Apple created a powerful development environment for application development on Mac OS X.

In 2007, Apple released an update to the Objective-C language and labeled it Objective-C 2.0. That version of the language formed the basis for the second edition of the book.

When the iPhone was released in 2007, developers clamored for the opportunity to develop applications for this revolutionary device. At first, Apple did not welcome third-party application development. The company's way of placating wannabe iPhone developers was to allow them to develop web-based applications. A web-based application runs under the iPhone's built-in Safari web browser and requires the user to connect to the website that hosts the application in order to run it. Developers were not satisfied with the many inherent limitations of web-based applications, and Apple shortly thereafter announced that developers would be able to develop so-called *native* applications for the iPhone.

A native application is one that resides on the iPhone and runs under the iPhone's operating system, in the same way that the iPhone's built-in applications (such as Contacts, Stocks, and Weather) run on the device. The iPhone's OS is actually a version of Mac OS X, which meant that applications could be developed and debugged on a MacBook Pro, for example. In fact, Apple soon provided a powerful Software Development Kit (SDK) that allowed for rapid iPhone application development and debugging. The availability of an iPhone simulator made it possible for developers to debug their applications directly on their development system, obviating the need to download and test the program on an actual iPhone or iPod Touch device.

With the introduction of the iPad in 2010, Apple started to genericize the terminology used for the operating system and the SDK that now support different devices with different physical sizes and screen resolutions. The iOS SDK allows you to develop applications for any iOS device and as of this writing, iOS 5 is the current release of the operating system.

What You Will Learn from This Book

When I contemplated writing a tutorial on Objective-C, I had to make a fundamental decision. As with other texts on Objective-C, I could write mine to assume that the reader already knew how to write C programs. I could also teach the language from the perspective of using the rich library of routines, such as the Foundation and UIKit frameworks. Some texts also take the approach of teaching how to use the development tools, such as the Mac's Xcode and the tool formerly known as Interface Builder to design the UI.

I had several problems adopting this approach. First, learning the entire C language before learning Objective-C is wrong. C is a *procedural* language containing many features that are not necessary for programming in Objective-C, especially at the novice level. In fact, resorting to some of these features goes against the grain of adhering to a good object-oriented programming methodology. It's also not a good idea to learn all the details of a procedural language before learning an object-oriented one. This starts the programmer in the wrong direction, and gives the wrong orientation and mindset for fostering a good object-oriented programming style. Just because Objective-C is an extension to the C language doesn't mean you have to learn C first.

So I decided neither to teach C first nor to assume prior knowledge of the language. Instead, I decided to take the unconventional approach of teaching Objective-C and the underlying C language as a single integrated language, from an object-oriented programming perspective. The purpose of this book is as its name implies: to teach you how to program in Objective-C. It does not profess to teach you in detail how to use the development tools that are available for entering and debugging programs, or to provide in-depth instructions on how to develop interactive graphical applications. You can learn all that material in greater detail elsewhere, after you've learned how to write programs in Objective-C. In fact, mastering that material will be much easier when you have a solid foundation of how to program in Objective-C. This book does not assume much, if any, previous programming experience. In fact, if you're a novice programmer, with some dedication and hard work you should be able to learn Objective-C as your first programming language. Other readers have been successful at this, based on the feedback I've received from the previous editions of this book.

This book teaches Objective-C by example. As I present each new feature of the language, I usually provide a small complete program example to illustrate the feature. Just as a picture is worth a thousand words, so is a properly chosen program example. You are strongly encouraged to run each program (all of which are available online) and compare the results obtained on your system to those shown in the text. By doing so, you will learn the language and its syntax, but you will also become familiar with the process of compiling and running Objective-C programs.

How This Book Is Organized

This book is divided into three logical parts. Part I, "The Objective-C Language," teaches the essentials of the language. Part II, "The Foundation Framework," teaches how to use the rich assortment of predefined classes that form the Foundation framework. Part III, "Cocoa, Cocoa Touch, and the iOS SDK," gives you an overview of the Cocoa and Cocoa Touch frameworks and then walks you through the process of developing a simple iOS application using the iOS SDK.

A *framework* is a set of classes and routines that have been logically grouped together to make developing programs easier. Much of the power of programming in Objective-C rests on the extensive frameworks that are available.

Chapter 2, "Programming in Objective-C," begins by teaching you how to write your first program in Objective-C.

Because this is not a book on Cocoa or iOS programming, graphical user interfaces (GUIs) are not extensively taught and are hardly even mentioned until Part III. So an approach was needed to get input into a program and produce output. Most of the examples in this text take input from the keyboard and produce their output in a window pane: a Terminal window if you're using the command line, or a debug output pane if you're using Xcode.

Chapter 3, "Classes, Objects, and Methods," covers the fundamentals of object-oriented programming. This chapter introduces some terminology, but it's kept to a minimum. I also introduce the mechanism for defining a class and the means for sending messages to instances or objects. Instructors and seasoned Objective-C programmers will notice that I use *static* typing for declaring objects. I think this is the best way for the student to get started because the compiler can catch more errors, making the programs more self-documenting and encouraging the new programmer to explicitly declare the data types when they are known. As a result, the notion of the `id` type and its power is not fully explored until Chapter 9, "Polymorphism, Dynamic Typing, and Dynamic Binding."

Chapter 4, "Data Types and Expressions," describes the basic Objective-C data types and how to use them in your programs.

Chapter 5, "Program Looping," introduces the three looping statements you can use in your programs: `for`, `while`, and `do`.

Making decisions is fundamental to any computer programming language. Chapter 6, "Making Decisions," covers the Objective-C language's `if` and `switch` statements in detail.

Chapter 7, "More on Classes," delves more deeply into working with classes and objects. Details about methods, multiple arguments to methods, and local variables are discussed here.

Chapter 8, "Inheritance," introduces the key concept of inheritance. This feature makes the development of programs easier because you can take advantage of what comes from above. Inheritance and the notion of subclasses make modifying and extending existing class definitions easy.

Chapter 9 discusses three fundamental characteristics of the Objective-C language. Polymorphism, dynamic typing, and dynamic binding are the key concepts covered here.

Chapters 10–13 round out the discussion of the Objective-C language, covering issues such as initialization of objects, blocks, protocols, categories, the preprocessor, and some of the underlying C features, including functions, arrays, structures, and pointers. These underlying features are often unnecessary (and often best avoided) when first developing object-oriented applications. It's recommended that you skim Chapter 13, "Underlying C Language Features," the first time through the text and return to it only as necessary to learn more about a particular feature of the language. Chapter 13 also introduces a recent addition to the C language known as *blocks*. This should be learned after you learn about how to write functions, since the syntax of the former is derived from the latter.

Part II begins with Chapter 14, "Introduction to the Foundation Framework," which gives an introduction to the Foundation framework and how to use its voluminous documentation.

Chapters 15–19 cover important features of the Foundation framework. These include number and string objects, collections, the file system, memory management, and the process of copying and archiving objects.

By the time you're done with Part II, you will be able to develop fairly sophisticated programs in Objective-C that work with the Foundation framework.

Part III starts with Chapter 20, "Introduction to Cocoa and Cocoa Touch" Here you'll get a quick overview of the frameworks that provide the classes you need to develop sophisticated graphical applications on the Mac and on your iOS devices.

Chapter 21, "Writing iOS Applications," introduces the iOS SDK and the UIKit framework. This chapter illustrates a step-by-step approach to writing a simple iOS application, followed by a more sophisticated calculator application that enables you to use your iPhone to perform simple arithmetic calculations with fractions.

Because object-oriented parlance involves a fair amount of terminology, Appendix A, "Glossary," provides definitions of some common terms.

Appendix B, "Address Book Example Source Code," gives the source code listing for two classes that are developed and used extensively in Part II of this text. These classes define address card and address book classes. Methods enable you to perform simple operations such as adding and removing address cards from the address book, looking up someone, listing the contents of the address book, and so on.

After you've learned how to write Objective-C programs, you can go in several directions. You might want to learn more about the underlying C programming language—or you might want to start writing Cocoa programs to run on Mac OS X, or develop more sophisticated iOS applications.

Support

If you go to classroomM.com/objective-c, you'll find a forum rich with content. There you can get source code (note that you won't find the "official" source code for all the examples there, as I am a firm believer that a big part the learning process occurs when you type in the program examples yourself and learn how to identify and correct any errors.), answers to exercises, errata, quizzes, and pose questions to me and fellow forum members. The forum has turned into a rich community of active members who are happy to help other members solve their problems and answer their questions. Please go, join, and participate!

Acknowledgments

I would like to acknowledge several people for their help in the preparation of the first edition of this text. First, I want to thank Tony Iannino and Steven Levy for reviewing the manuscript. I am also grateful to Mike Gaines for providing his input.

I'd also like to thank my technical editors, Jack Purdum (first edition) and Mike Trent. I was lucky enough to have Mike review the first two editions of this text. He provided the most thorough review of any book I've ever written. Not only did he point out weaknesses, but he was also generous enough to offer his suggestions. Because of Mike's comments in the first edition, I changed my approach to teaching memory management and tried to make sure that every program example in this book was "leak free." This was prior to the fourth edition, where the strong emphasis on memory management became obsolete with the introduction of ARC. Mike also provided invaluable input for my chapter on iPhone programming.

From the first edition, Catherine Babin supplied the cover photograph and provided me with many wonderful pictures to choose from. Having the cover art from a friend made the book even more special.

I am so grateful to Mark Taber (for all editions) from Pearson for putting up with all delays and for being kind enough to work around my schedule and to tolerate my consistent missing of deadlines. I am extremely grateful to Michael de Haan and Wendy Mui for doing an incredible, unsolicited job proofreading the second edition (and thanks Wendy for your work on the third edition as well). Their meticulous attention to detail has resulted in a list of both typographical and substantive errors that have been addressed in the second printing. Publishers take note: These two pairs of eyes are priceless!

As noted at the start of this Introduction, Dennis Ritchie invented the C language. He was also a co-inventor of the Unix operating system, which is the basis for Mac OS X and iOS. Sadly, the world lost both Dennis Ritchie and Steve Jobs within the span of a week. These two people had a profound effect on my career. Needless to say, this book would not exist if not for them.

Finally, I'd like to thank the members of the forum at classroomM.com/objective-c for all their feedback, support, and kind words.

Preface to the Fourth Edition

When I attended Apple's World Wide Developer's Conference (WWDC) in June 2011, I was in for quite a surprise. The third edition of this book had been written and was scheduled for release in just a few short weeks. What Apple announced there with respect to Objective-C was a game-changer for new, would-be Objective-C programmers. Prior to Xcode 4.2 (and the Apple LLVM 3.0 compiler it contained), iOS developers had to struggle with the perils of memory management, which involved judiciously tracking objects and telling the system when to hold onto and when to release them. Making the smallest mistake in this could and did easily cause applications to crash. Well, at WWDC 2011 Apple introduced a new version of the Objective-C compiler that contained a feature called ARC, which is short for Automatic Reference Counting. With ARC, programmers no longer needed to worry about their object's life cycle; the compiler handles it all automatically for them!

I must apologize for such a short period of time between editions, but this fundamental change in how to approach teaching the language made this fourth edition necessary. So this edition assumes you're using Xcode 4.2 or later and that you're using ARC. If you're not, you need to still learn about manual memory management, which is briefly covered in Chapter 17, "Memory Management and Automatic Reference Counting."

Stephen G. Kochan
October 2011

2

Programming in Objective-C

In this chapter, we dive right in and show you how to write your first Objective-C program. You won't work with objects just yet; that's the topic of the next chapter. We want you to understand the steps involved in keying in a program and compiling and running it.

To begin, let's pick a rather simple example: a program that displays the phrase "Programming is fun!" on your screen. Without further ado, Program 2.1 shows an Objective-C program to accomplish this task.

Program 2.1

```
// First program example

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
   @autoreleasepool {
      NSLog (@"Programming is fun!");
   }
   return 0;
}
```

Compiling and Running Programs

Before we go into a detailed explanation of this program, we need to cover the steps involved in compiling and running it. You can both compile and run your program using Xcode, or you can use the Clang Objective-C compiler in a Terminal window. Let's go through the sequence of steps using both methods. Then you can decide how you want to work with your programs throughout the rest of this book.

> **Note**
>
> You'll want to go to developer.apple.com and make sure you have the latest version of the Xcode development tools. There you can download Xcode and the iOS SDK at no charge. If you're not a registered developer, you'll have to register first. That can also be done at no charge. Note that Xcode is also available for a minimal cost from the Mac App Store.

Using Xcode

Xcode is a sophisticated application that enables you to easily type in, compile, debug, and execute programs. If you plan on doing serious application development on the Mac, learning how to use this powerful tool is worthwhile. We just get you started here. Later we return to Xcode and take you through the steps involved in developing a graphical application with it.

> **Note**
>
> As mentioned, Xcode is a sophisticated tool, and the introduction of Xcode 4 added even more features. It's easy to get lost using this tool. If that happens to you, back up a little and try reading the Xcode User Guide, which can be accessed from Xcode help menu, to get your bearings.

Xcode is located in the `Developer` folder inside a subfolder called `Applications`. Figure 2.1 shows its icon.

Figure 2.1 Xcode icon

Start Xcode. You can then select "Create a New Xcode Project" from the startup screen. Alternatively, under the File menu, select New, New Project… (see Figure 2.2).

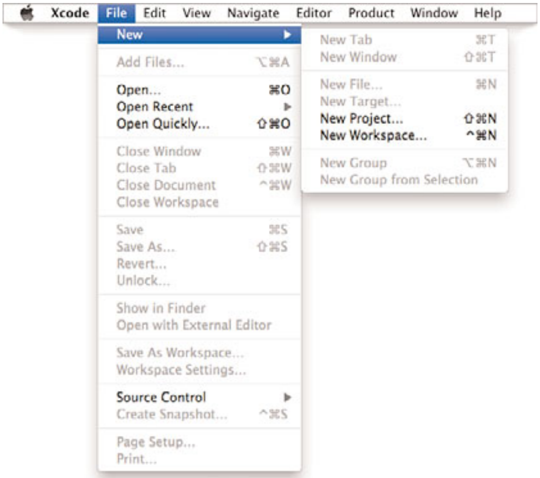

Figure 2.2 Starting a new project

A window appears, as shown in Figure 2.3.

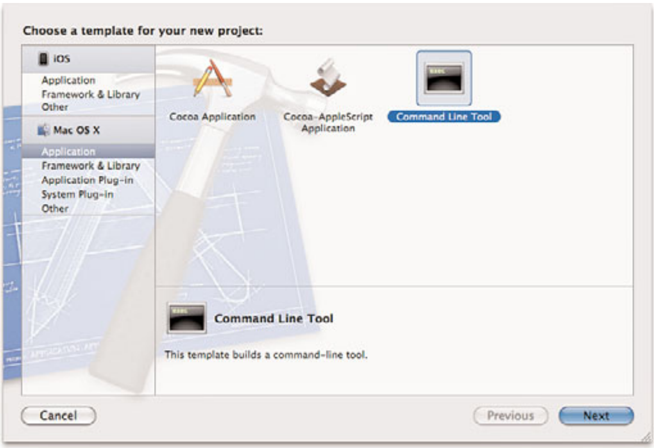

Figure 2.3 Starting a new project: selecting the application type

In the left pane, you'll see a section labeled Mac OS X. Select Application. In the upper-right pane, select Command Line Tool, as depicted in the previous figure. On the next pane that appears, you pick your application's name. Enter prog1 for the Product Name and make sure Foundation is selected for the Type. Also, be sure that the Use Automatic Reference Counting box is checked. Your screen should look like Figure 2.4.

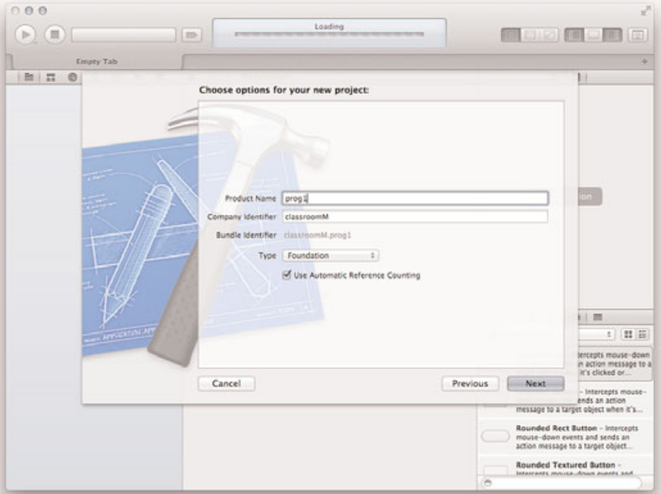

Figure 2.4 Starting a new project: specifying the product name and type

Click Next. The dropdown that appears allows you to specify the name of the project folder that will contain the files related to your project. Here, you can also specify where you want that project folder stored. According to Figure 2.5 we're going to store our project on the Desktop in a folder called prog1.

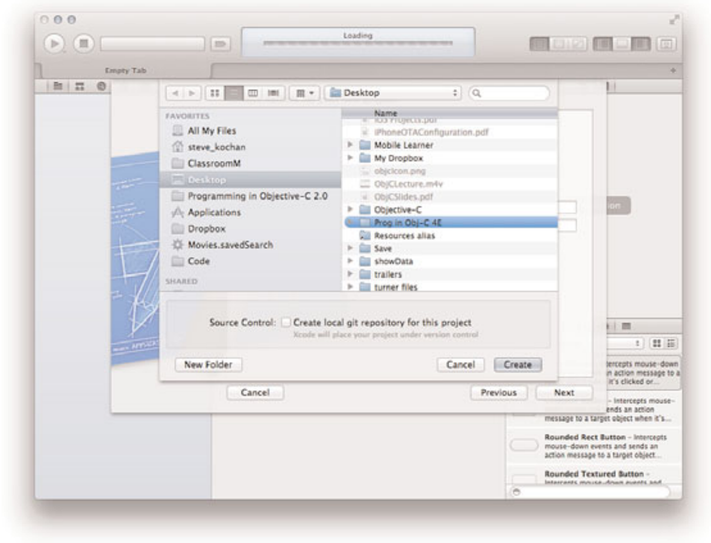

Figure 2.5 Selecting the location and name of the project folder

Click the Create button to create your new project. Xcode will open a project window such as the one shown in Figure 2.6. Note that your window might look different if you've used Xcode before or have changed any of its options.

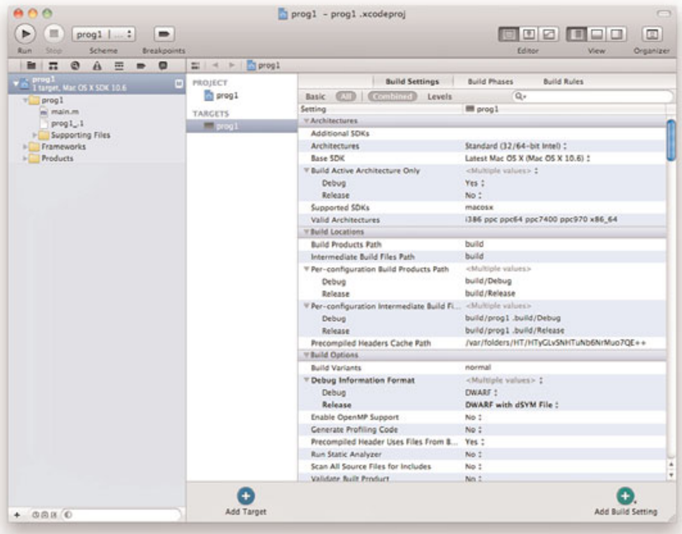

Figure 2.6 Xcode **prog1** project window

Now it's time to type in your first program. Select the file `main.m` in the left pane (you may have to reveal the files under the project name by clicking the disclosure triangle). Your Xcode window should now appear as shown in Figure 2.7.

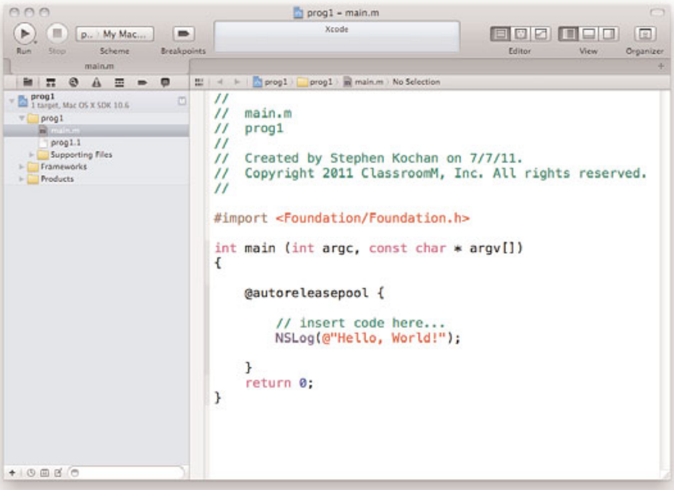

Figure 2.7 File **`main.m`** and edit window

Objective-C source files use `.m` as the last two characters of the filename (known as its *extension*). Table 2.1 lists other commonly used filename extensions.

Table 2.1 **Common Filename Extensions**

Extension	Meaning
`.c`	C language source file
`.cc, .cpp`	C++ language source file
`.h`	Header file
`.m`	Objective-C source file
`.mm`	Objective-C++ source file
`.pl`	Perl source file
`.o`	Object (compiled) file

Returning to your Xcode project window, the right pane shows the contents of the file called `main.m`, which was automatically created for you as a template file by Xcode, and which contains the following lines:

```
//
//  main.m
//  prog1
//
//  Created by Steve Kochan on 7/7/11.
//  Copyright 2011 ClassroomM, Inc.. All rights reserved.
//
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
   @autoreleasepool {

      // insert code here...
      NSLog (@"Hello World!");
   }
   return 0;
}
```

You can edit your file inside this window. Make changes to the program shown in the Edit window to match Program 2.1. The lines that start with two slash characters (`//`) are called *comments*; we talk more about comments shortly.

Your program in the edit window should now look like this (don't worry if your comments don't match).

Program 2.1

```
// First program example

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
```

```
    @autoreleasepool {
        NSLog (@"Programming is fun!");
    }
    return 0;
}
```

> **Note**
>
> Don't worry about all the colors shown for your text onscreen. Xcode indicates values, reserved words, and so on with different colors. This will prove very valuable as you start programming more, as it can indicate the source of a potential error.

Now it's time to compile and run your first program—in Xcode terminology, it's called *building and running*. Before doing that, we need to reveal a window pane that will display the results (output) from our program. You can do this most easily by selecting the middle icon under View in the toolbar. When you hover over this icon, it says "Hide or show the Debug area." Your window should now appear as shown in Figure 2.8. Note that XCode will normally reveal the Debug area automatically whenever any data is written to it.

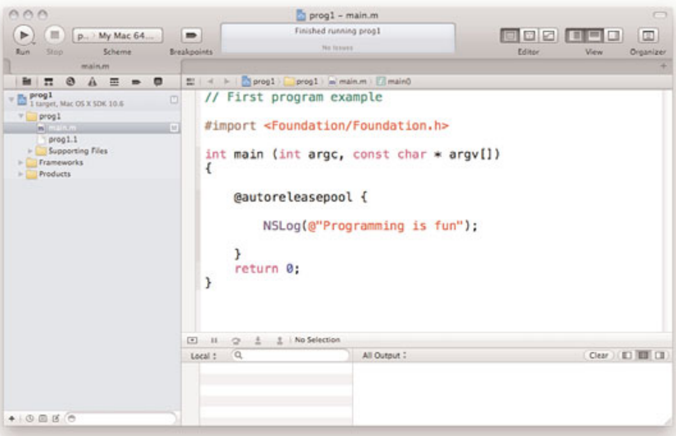

Figure 2.8 Xcode Debug area revealed

Now, if you press the Run button located at the top left of the toolbar or select Run from the Product menu, Xcode will go through the two-step process of first building and then running your program. The latter occurs only if no errors are discovered in your program.

If you do make mistakes in your program, along the way you'll see errors denoted as red stop signs containing exclamation points—these are known as *fatal errors* and you can't run your program without correcting these. *Warnings* are depicted by yellow triangles containing exclamation points—you can still run your program with them, but in general you should examine and correct them. After running the program with all the errors removed, the lower right pane will display the output from your program and should look similar to Figure 2.9. Don't worry about the verbose messages that appear. The output line we're interested in is the one you see in bold.

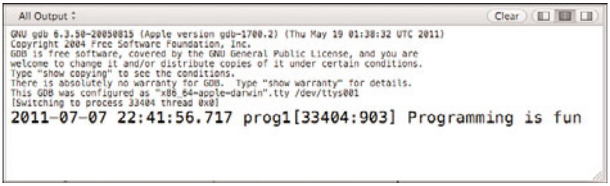

Figure 2.9 Xcode Debug output

You're now done with the procedural part of compiling and running your first program with Xcode (whew!). The following summarizes the steps involved in creating a new program with Xcode:

1. Start the Xcode application.
2. If this is a new project, select File, New, New Project... or choose Create a New Xcode Project from the startup screen.
3. For the type of application, select Application, Command Line Tool, and click Next.
4. Select a name for your application and set its Type to Foundation. Make sure Use Automatic Reference Counting is checked. Click Next.
5. Select a name for your project folder, and a directory to store your project files in. Click Create.
6. In the left pane, you will see the file `main.m` (you might need to reveal it from inside the folder that has the product's name). Highlight that file. Type your program into the edit window that appears in the rightmost pane.
7. In the toolbar, select the middle icon under View. This will reveal the Debug area. That's where you'll see your output.
8. Build and run your application by clicking the Run button in the toolbar or selecting Run from the Product menu.

> **Note**
>
> Xcode contains a powerful built-in tool known as the static analyzer. It does an analysis of your code and can find program logic errors. You can use it by selecting Analyze from the Product menu or from the Run button in the toolbar.

9. If you get any compiler errors or the output is not what you expected, make your changes to the program and rerun it.

Using Terminal

Some people might want to avoid having to learn Xcode to get started programming with Objective-C. If you're used to using the UNIX shell and command-line tools, you might want to edit, compile, and run your programs using the Terminal application. Here, we examine how to go about doing that.

The first step is to start the Terminal application on your Mac. The Terminal application is located in the Applications folder, stored under Utilities. Figure 2.10 shows its icon.

Figure 2.10 Terminal program icon

Start the Terminal application. You'll see a window that looks like Figure 2.11.

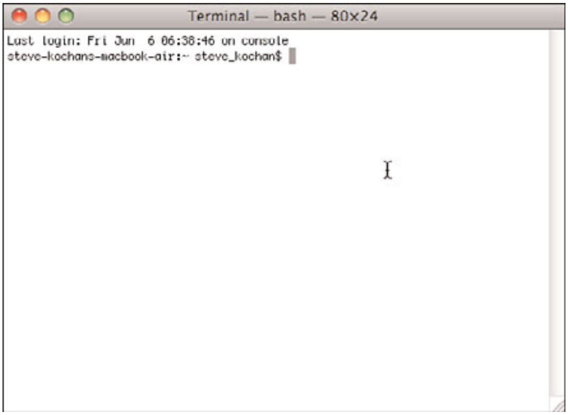

Figure 2.11 Terminal window

You type commands after the `$` (or `%`, depending on how your Terminal application is configured) on each line. If you're familiar with using UNIX, you'll find this straightforward.

First, you need to enter the lines from Program 2.1 into a file. You can begin by creating a directory in which to store your program examples. Then, you must run a text editor, such as vi or emacs, to enter your program:

```
sh-2.05a$ mkdir Progs    Create a directory to store programs in
sh-2.05a$ cd Progs       Change to the new directory
sh-2.05a$ vi main.m      Start up a text editor to enter program
 --
```

> **Note**
> In the previous example and throughout the remainder of this text, commands that you, the user, enter are indicated in boldface.

For Objective-C files, you can choose any name you want; just make sure the last two characters are `.m`. This indicates to the compiler that you have an Objective-C program.

After you've entered your program into a file (and we're not showing the edit commands to enter and save your text here), you can use the LLVM Clang Objective-C compiler, which is called clang, to compile and link your program. This is the general format of the `clang` command:

```
clang -fobjc-arc -framework Foundation files -o program
```

This option says to use information about the Foundation framework:

```
-framework Foundation
```

Just remember to use this option on your command line. *`files`* is the list of files to be compiled. In our example, we have only one such file, and we're calling it `main.m`. *`progname`* is the name of the file that will contain the executable if the program compiles without any errors.

We'll call the program `prog1`; here, then, is the command line to compile your first Objective-C program:

```
$ clang -fobjc-arc –framework Foundation main.m -o prog1  Compile main.m & call it prog1
$
```

The return of the command prompt without any messages means that no errors were found in the program. Now you can subsequently execute the program by typing the name prog1 at the command prompt:

```
$ prog1        Execute prog1
sh: prog1: command not found
$
```

This is the result you'll probably get unless you've used Terminal before. The UNIX shell (which is the application running your program) doesn't know where `prog1` is located (we don't go into all the details of this here), so you have two options: One is to precede the name of the program with the characters `./` so that the shell knows to look in the current directory for the program to execute. The other is to add the directory in which your programs are stored (or just simply the current directory) to the shell's PATH variable. Let's take the first approach here:

```
$ ./prog1        Execute prog1
2008-06-08 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

You should note that writing and debugging Objective-C programs from the terminal is a valid approach. However, it's not a good long-term strategy. If you want to build Mac OS X or iOS applications, there's more to just the executable file that needs to be "packaged" into an application bundle. It's not easy to do that from the Terminal application, and it's one of Xcode's specialties. Therefore, I suggest you start learning to use Xcode to develop your programs. There is a learning curve to do this, but the effort will be well worth it in the end.

Explanation of Your First Program

Now that you are familiar with the steps involved in compiling and running Objective-C programs, let's take a closer look at this first program. Here it is again:

```
//
//  main.m
//  prog1
//
//  Created by Steve Kochan on 7/7/11.
//  Copyright 2011 ClassroomM, Inc.. All rights reserved.
//

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
   @autoreleasepool {

      NSLog (@"Programming is fun!");

   }
   return 0;
}
```

In Objective-C, lowercase and uppercase letters are distinct. Also, Objective-C doesn't care where on the line you begin typing—you can begin typing your statement at any position on the line. You can use this to your advantage in developing programs that are easier to read.

The first seven lines of the program introduce the concept of the *comment*. A comment statement is used in a program to document a program and enhance its readability. Comments tell the reader of the program—whether it's the programmer or someone else whose responsibility it is to maintain the program—just what the programmer had in mind when writing a particular program or a particular sequence of statements.

You can insert comments into an Objective-C program in two ways. One is by using two consecutive slash characters (`//`). The compiler ignores any characters that follow these slashes, up to the end of the line.

You can also initiate a comment with the two characters `/` and `*`. This marks the beginning of the comment. These types of comments have to be terminated. To end the comment, you use the characters `*` and `/`, again without any embedded spaces. All characters included between the opening `/*` and the closing `*/` are treated as part of the comment statement and are ignored by the Objective-C compiler. This form of comment is often used when comments span many lines of code, as in the following:

```
/*
 This file implements a class called Fraction, which
 represents fractional numbers. Methods allow manipulation of
 fractions, such as addition, subtraction, etc.

 For more information, consult the document:
    /usr/docs/classes/fractions.pdf
*/
```

Which style of comment you use is entirely up to you. Just note that you can't nest the `/*` style comments.

Get into the habit of inserting comment statements in the program as you write it or type it into the computer, for three good reasons. First, documenting the program while the particular program logic is still fresh in your mind is far easier than going back and rethinking the logic after the program has been completed. Second, by inserting comments into the program at such an early stage of the game, you can reap the benefits of the comments during the debug phase, when program logic errors are isolated and debugged. Not only can a comment help you (and others) read through the program, but it also can help point the way to the source of the logic mistake. Finally, I haven't yet discovered a programmer who actually enjoys documenting a program. In fact, after you've finished debugging your program, you will probably not relish the idea of going back to the program to insert comments. Inserting comments while developing the program makes this sometimes-tedious task a bit easier to handle.

This next line of Program 2.1 tells the compiler to locate and process a file named `Foundation.h`:

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

This is a system file—that is, not a file that you created. `#import` says to import or include the information from that file into the program, exactly as if the contents of the file were typed into the program at that point. You imported the file `Foundation.h` because it has information about other classes and functions that are used later in the program.

In Program 2.1, this line specifies that the name of the program is `main`:

```
int main (int argc, const char * argv[])
```

`main` is a special name that indicates precisely where the program is to begin execution. The reserved word `int` that precedes main specifies the type of value main returns, which is an integer (more about that soon). We ignore what appears between the open and closed parentheses for now; these have to do with *command-line arguments*, a topic we address in Chapter 13, "Underlying C Language Features."

Now that you have identified `main` to the system, you are ready to specify precisely what this routine is to perform. This is done by enclosing all the program *statements* of the routine within a pair of curly braces. In the simplest case, a statement is just an expression that is terminated with a semicolon. The system treats all the program statements included between the braces as part of the `main` routine.

The next line in `main` reads

```
@autoreleasepool {
```

Any program statements between the `{` and the matching closing `}` are executed within a context known an *autorelease pool*. The autorelease pool is a mechanism that allows the system to efficiently manage the memory your application uses as it creates new objects. I mention it in more detail in Chapter 17, "Memory Management and Automatic Reference Counting." Here, we have one statement inside our `@autoreleasepool` context.

That statement specifies that a routine named `NSLog` is to be invoked, or *called*. The parameter, or *argument*, to be passed or handed to the `NSLog` routine is the following string of characters:

```
@"Programming is fun!"
```

Here, the `@` sign immediately precedes a string of characters enclosed in a pair of double quotes. Collectively, this is known as a constant `NSString` object.

> **Note**
>
> If you have C programming experience, you might be puzzled by the leading `@` character. Without that leading `@` character, you are writing a constant C-style string; with it, you are writing an `NSString` string object. More on this topic in Chapter 15.

The `NSLog` routine is a function in the Objective-C library that simply displays or logs its argument (or arguments, as you will see shortly). Before doing so, however, it displays the date and time the routine is executed, the program name, and some other numbers we don't describe here. Throughout the rest of this book, we don't bother to show this text that `NSLog` inserts before your output.

You must terminate all program statements in Objective-C with a semicolon (`;`). This is why a semicolon appears immediately after the closed parenthesis of the `NSLog` call.

The final program statement in `main` looks like this:

```
return 0;
```

It says to terminate execution of `main` and to send back, or *return*, a status value of `0`. By convention, `0` means that the program ended normally. Any nonzero value typically means some problem occurred—for example, perhaps the program couldn't locate a file that it needed.

If you're using Xcode and you glance back to your output window (refer to Figure 2.9), you'll recall that the following displayed after the line of output from `NSLog`:

```
Program exited with status value:0.
```

You should understand what that message means now.

Now that we have finished discussing your first program, let's modify it to also display the phrase "And programming in Objective-C is even more fun!" You can do this by simply adding another call to the `NSLog` routine, as shown in Program 2.2. Remember that every Objective-C program statement must be terminated by a semicolon. Note that we've removed the leading comment lines in all the following program examples.

Program 2.2

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
       NSLog (@"Programming is fun!");
       NSLog (@"Programming in Objective-C is even more fun!");
    }
    return 0;
}
```

If you type in Program 2.2 and then compile and execute it, you can expect the following output (again, without showing the text that `NSLog` normally prepends to the output):

Program 2.2 Output

```
Programming is fun!
Programming in Objective-C is even more fun!
```

As you will see from the next program example, you don't need to make a separate call to the `NSLog` routine for each line of output.

First, let's talk about a special two-character sequence. The backslash (`\`) and the letter `n` are known collectively as the *newline* character. A newline character tells the system to do precisely what its name implies: go to a new line. Any characters to be printed after the newline character then appear on the next line of the display. In fact, the newline character is very similar in concept to the carriage return key on a typewriter (remember those?).

Study the program listed in Program 2.3 and try to predict the results before you examine the output (no cheating, now!).

Program 2.3

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

int main (int argc, const char *argv[])
{
   @autoreleasepool {

      NSLog (@"Testing...\n..1\n...2\n....3");
   }
   return 0;
}
```

Program 2.3 Output

```
Testing...
..1
...2
....3
```

Displaying the Values of Variables

Not only can simple phrases be displayed with `NSLog`, but the values of variables and the results of computations can be displayed as well. Program 2.4 uses the `NSLog` routine to display the results of adding two numbers, 50 and 25.

Program 2.4

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

int main (int argc, const char *argv[])
{
   @autoreleasepool {

      int sum;

      sum = 50 + 25;
      NSLog (@"The sum of 50 and 25 is %i", sum);
   }

   return 0;
}
```

Program 2.4 Output

```
The sum of 50 and 25 is 75
```

The first program statement inside `main` after the autorelease pool is set up defines the variable `sum` to be of type `integer`. You must define all program variables before you can use them in a program. The definition of a variable specifies to the Objective-C compiler how the program should use it. The compiler needs this information to generate the correct instructions to store and retrieve values into and out of the variable. A variable defined as type `int` can be used to hold only integral values—that is, values without decimal places. Examples of integral values are `3`, `5`, `-20`, and `0`. Numbers with decimal places, such as `2.14`, `2.455`, and `27.0`, are known as *floating-point* numbers and are real numbers.

The integer variable sum stores the result of the addition of the two integers 50 and 25. We have intentionally left a blank line following the definition of this variable to visually separate the variable declarations of the routine from the program statements; this is strictly a matter of style. Sometimes adding a single blank line in a program can make the program more readable.

The program statement reads as it would in most other programming languages:

```
sum = 50 + 25;
```

The number `50` is added (as indicated by the plus sign) to the number `25`, and the result is stored (as indicated by the assignment operator, the equals sign) in the variable `sum`.

The `NSLog` routine call in Program 2.4 now has two arguments enclosed within the parentheses. These arguments are separated by a comma. The first argument to the `NSLog` routine is always the character string to be displayed. However, along with the display of the character string, you often want to have the value of certain program variables displayed as well. In this case, you want to have the value of the variable `sum` displayed after these characters are displayed:

```
The sum of 50 and 25 is
```

The percent character inside the first argument is a special character recognized by the `NSLog` function. The character that immediately follows the percent sign specifies what type of value is to be displayed at that point. In the previous program, the `NSLog` routine recognizes the letter `i` as signifying that an integer value is to be displayed.

Whenever the `NSLog` routine finds the `%i` characters inside a character string, it automatically displays the value of the next argument to the routine. Because `sum` is the next argument to `NSLog`, its value is automatically displayed after "The sum of 50 and 25 is".

Now try to predict the output from Program 2.5.

Program 2.5

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

int main (int argc, const char *argv[])
{
   @autoreleasepool {
      int value1, value2, sum;

      value1 = 50;
      value2 = 25;
      sum = value1 + value2;

      NSLog (@"The sum of %i and %i is %i", value1, value2, sum);
```

```
    }
    return 0;
}
```

Program 2.5 Output

```
The sum of 50 and 25 is 75
```

The second program statement inside `main` defines three variables called `value1`, `value2,` and `sum`, all of type `int`. This statement could have equivalently been expressed using three separate statements, as follows:

```
int value1;
int value2;
int sum;
```

After the three variables have been defined, the program assigns the value 50 to the variable `value1` and then the value `25` to `value2`. The sum of these two variables is then computed and the result assigned to the variable `sum`.

The call to the `NSLog` routine now contains four arguments. Once again, the first argument, commonly called the `format string`, describes to the system how the remaining arguments are to be displayed. The value of `value1` is to be displayed immediately following the phrase "The sum of." Similarly, the values of `value2` and `sum` are to be printed at the points indicated by the next two occurrences of the `%i` characters in the format string.

Summary

After reading this introductory chapter on developing programs in Objective-C, you should have a good feel of what is involved in writing a program in Objective-C—and you should be able to develop a small program on your own. In the next chapter, you begin to examine some of the intricacies of this powerful and flexible programming language. But first, try your hand at the exercises that follow, to make sure you understand the concepts presented in this chapter.

Exercises

1. Type in and run the five programs presented in this chapter. Compare the output produced by each program with the output presented after each program.

2. Write a program that displays the following text:
   ```
   In Objective-C, lowercase letters are significant.
   main is where program execution begins.
   Open and closed braces enclose program statements in a routine.
   All program statements must be terminated by a semicolon.
   ```

3. What output would you expect from the following program?

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
   @autoreleasepool {
      int i;
      i = 1;
      NSLog (@"Testing...");
      NSLog (@"....%i", i);
      NSLog (@"...%i",  i + 1);
      NSLog (@"..%i", i + 2);
   }
   return 0;
}
```

4. Write a program that subtracts the value `15` from `87` and displays the result, together with an appropriate message.

5. Identify the syntactic errors in the following program. Then type in and run the corrected program to make sure you have identified all the mistakes:

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

int main (int argc, const char *argv[]);
(
   @autoreleasepool {
      INT sum;
      /* COMPUTE RESULT //
      sum = 25 + 37 - 19
      / DISPLAY RESULTS /
      NSLog (@'The answer is %i' sum);
   }
   return 0;
}
```

6. What output would you expect from the following program?

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

int main (int argc, const char *argv[])
{
   @autoreleasepool {
      int answer, result;

      answer = 100;
      result = answer - 10;

      NSLog (@"The result is %i\n", result + 5);
   }
   return 0;
}
```

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[])
{
   @autoreleasepool {
      int  numerator = 1;
      int  denominator = 3;
      NSLog (@"The fraction is %i/%i", numerator, denominator);
   }
   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 `@autoreleasepool` directive, 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

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

@end

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


@implementation Fraction
{
   int  numerator;
   int  denominator;
}
-(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[])
{
   @autoreleasepool {
      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];
   }
   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 and its methods, whereas the @implementation section describes the data (the *instance variables* that objects from the class will store) and contains the actual code that implements the methods declared in the interface section. Finally, the program section contains the program code to carry out the intended purpose of the program.

> **Note**
>
> You can also declare the instance variables for a class in the interface section. The ability to do it in the implementation section was added as of Xcode 4.2 and is considered a better way to define a class. You learn more about why in a later chapter.

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 tell the Objective-C compiler where the class came from. That is, you have to name its *parent* class. Next, you need to define the type of operations, or *methods*, that can be used when working with objects from this class. And, as you learn in a later chapter, you also list items known as *properties* in this special section of the program called the @interface section. The general format of this section looks like this:

```
@interface NewClassName: ParentClassName
   propertyAndMethodDeclarations;
@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

-(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`.

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.

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.

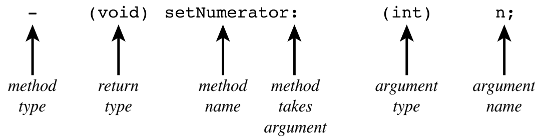

Figure 3.1 Declaring a method

The `@implementation` Section

As noted, the `@implementation` section contains the actual code for the methods you declared in the `@interface` section. 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*. 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
{
   memberDeclarations;
}
  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 *`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. 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. 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. It's noted here that you can have the compiler automatically generate methods for you by using a special `@synthesize` directive. This is covered in detail in Chapter 7.

Consider the `@implementation` section from Program 3.2:

```
//---- @implementation section ----
@implementation Fraction
{
   int  numerator;
   int  denominator;
}

-(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[])
{
   @autoreleasepool {
      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];
   }

   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];
```

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