Stephen G. Kochan

Updated for
Xcode 5
and
iOS 7

Programming in Objective-C

Sixth Edition

FREE SAMPLE CHAPTER

Programming in Objective-C

Sixth 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 978-0-321-83389-1

MySQL
Paul DuBois
ISBN-13: 978-0-321-83387-7

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 in print and in electronic formats at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

Developer’s Library
informit.com/devlibrary

Programming in Objective-C

Sixth 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, Sixth Edition

ISBN-13: 978-0-321-96760-2

ISBN-10: 0-321-96760-7

Library of Congress Control Number: 2013954275

Printed in the United States of America

First Printing: December 2013

Trademarks

Warning and Disclaimer

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Indexers
Erika Millen
Cheryl Lenser

Proofreader
Dan Knott

Technical Editor
Michael Trent

Publishing Coordinator
Vanessa Evans

Designer
Chuti Prasertsith

Compositor
Tricia Bronkella

❖

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

Appendixes

Table of Contents

About the Author

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

About the Technical Reviewers

Michael Trent has been programming in Objective-C since 1997—and programming Macs since well before that. He is a regular contributor to programming websites, 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's Cocoa frameworks to build professional video applications for Mac OS X. He 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.

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, she 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 Bachelor of Arts degree in mathematics from the University of California Berkeley.

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.

This page intentionally left blank

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**
>
> Xcode is available from the Mac App Store. However, you can also get pre-release versions of Xcode by becoming a registered Apple developer (no charge for that). Go to http://developer.apple.com to get the latest version of the Xcode development tools. There you can download Xcode and the iOS software development kit (SDK) for no charge.

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 5 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 you can access from the Xcode Help menu, to get your bearings.

Once installed, Xcode is in your Applications folder. Figure 2.1 shows its icon.

Figure 2.1 Xcode icon

Start Xcode. (The first time you launch the application, you have to go through some one-time things like agreeing to the license agreement.) You can then select Create a New Xcode Project from the startup screen (see Figure 2.2). Alternatively, under the File menu, select New, Project.

A window appears, as shown in Figure 2.3.

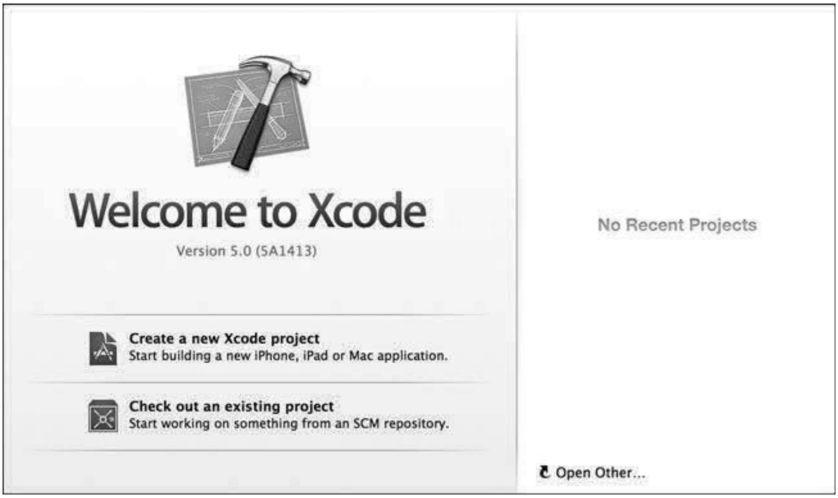

Figure 2.2 Starting a new project

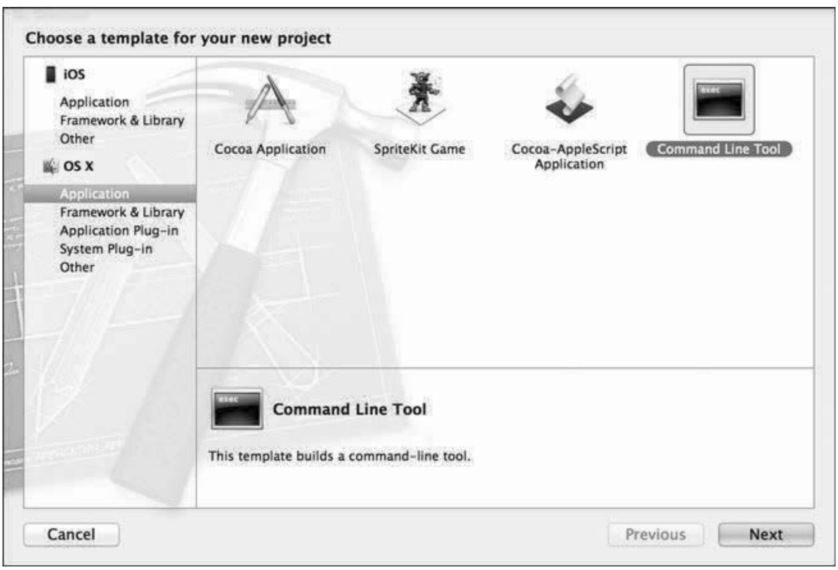

Figure 2.3 Starting a new project: selecting the application type

In the left pane, you'll see a section labeled 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 type in something in the Company Identifier and Bundle Identifier fields. The latter field is used for creating iOS apps, so we don't need to be too concerned at this point about what's entered there. Make sure Foundation is selected for the Type. Your screen should look like Figure 2.4.

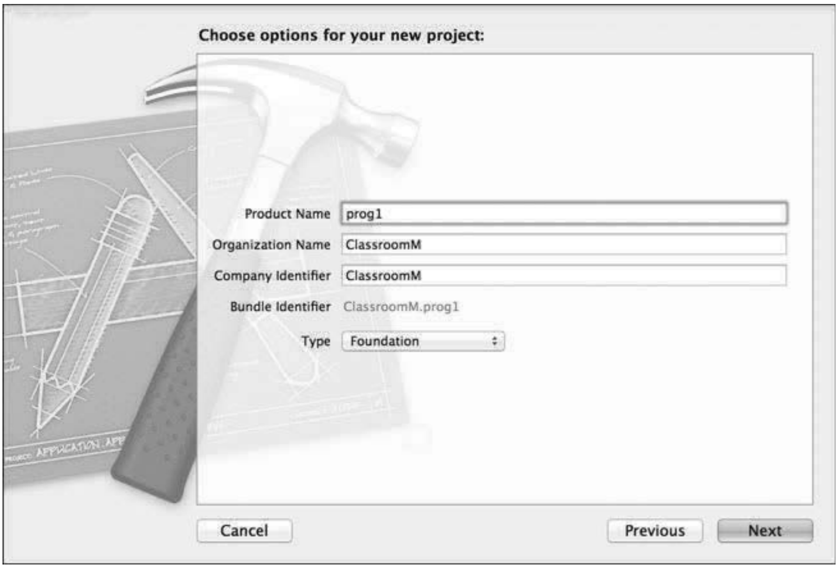

Figure 2.4 Starting a new project: specifying the product name and type

Click Next. On the sheet that appears, you can 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.

Click the Create button to create your new project. Xcode then opens 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. This figure shows the Utilities pane (the right-most pane). You can close that pane by deselecting the third icon listed in the *View* category in the top-right corner of your Xcode toolbar. Note that the categories are not labeled by default. To get the labels to appear, right click in the Toolbar and select Icon and Text.

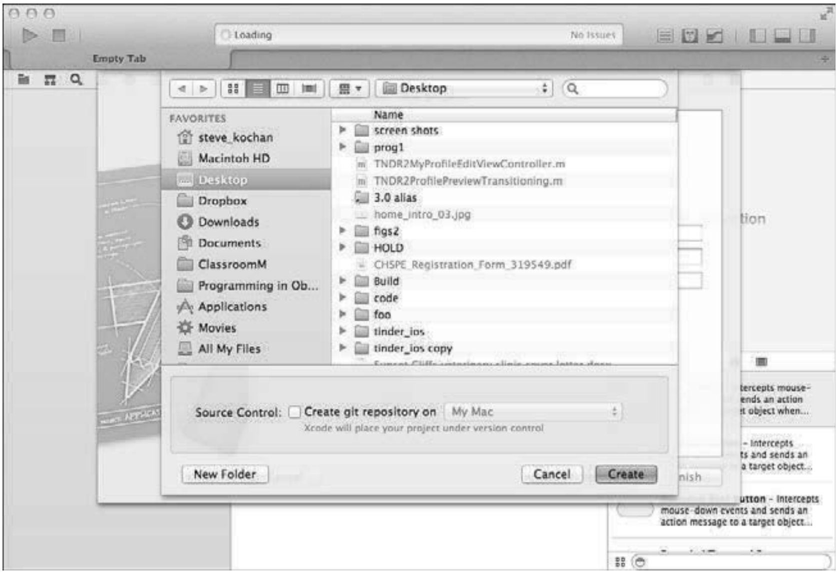

Figure 2.5 Selecting the location and name of the project folder

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 might have to reveal the files under the project name by clicking the disclosure triangle.) Your Xcode window should now look like Figure 2.7.

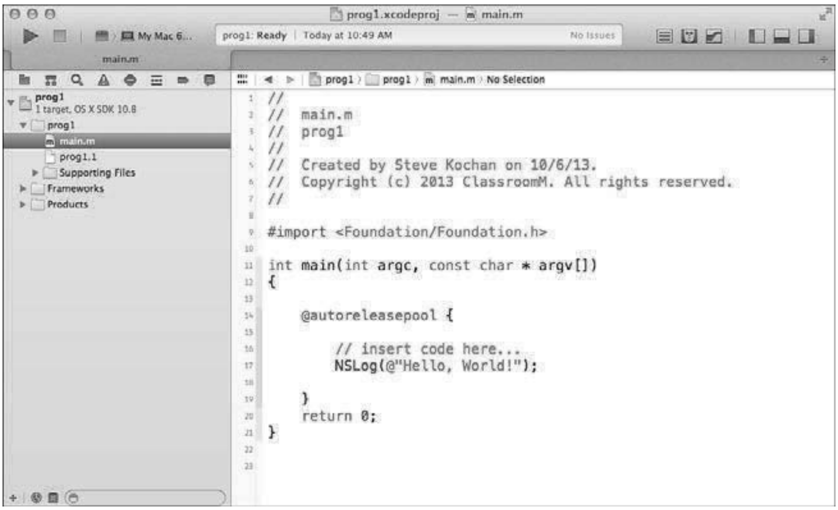

Figure 2.7 File `main.m` and the edit window

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

The right pane of your Xcode project window 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 10/16/13.
//  Copyright (c) 2013 ClassroomM. 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 pane that will display the results (output) from our program. You can do this most easily by selecting the middle icon in the "View" (rightmost) category on the toolbar. When you hover over this icon, it says Hide or Show the Debug Area. Your window should now look like Figure 2.8. Note that Xcode normally reveals the debug area automatically whenever any data is written to it.

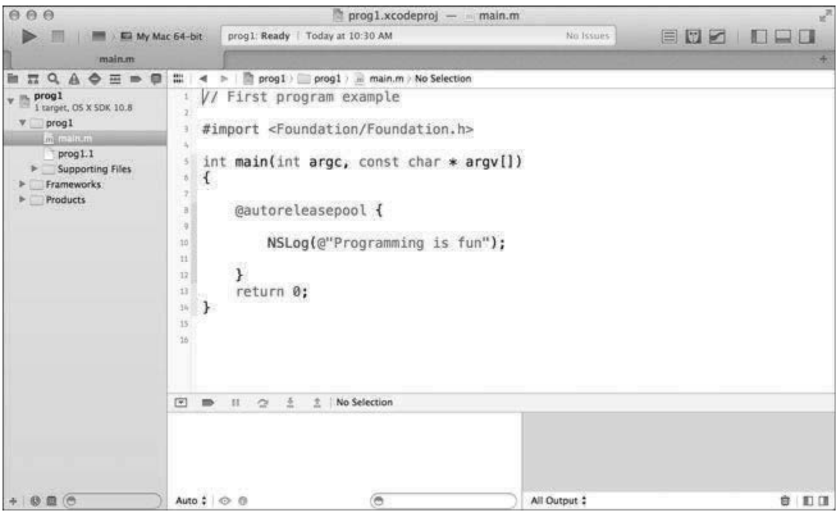

Figure 2.8 Xcode debug area revealed

Now, if you click the "Play" button located at the top left of the toolbar or select Run from the Product menu, Xcode goes 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.

> **Note**
>
> The first time you click the Run button Xcode displays a sheet reading Enable Developer Mode on the Mac? Click the Enable button and enter your admin password to proceed.

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 you run the program with all the errors removed, the lower-right pane displays the output from your program and should look similar to Figure 2.9.

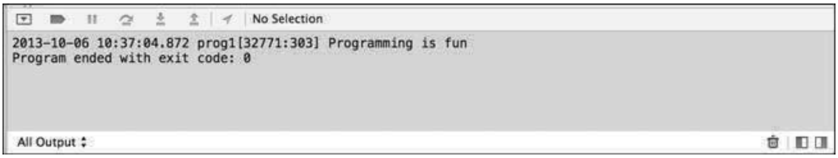

Figure 2.9 Xcode debug output

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, 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. Fill in the other fields that appear on the sheet. 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. On the toolbar, select the middle icon in the upper-right corner to reveal the debug area. That's where you'll see your output.
8. Build and run your application by clicking the Play button on 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 Play button on 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.

Before attempting to compile you program from the command line, make sure that you have Xcode's Command Line Tools installed on your system. Go to Xcode, Preferences, Downloads, Components from inside Xcode. You'll see something similar to Figure 2.10. This figure indicates that the Command Line Tools have not been installed on this system. If they haven't, an Install button will be shown, which you can click to install the tools.

Figure 2.10 Installing the Command Line Tools

Once the Command Line Tools have been installed, the next step is to start the Terminal application on your Mac. The Terminal application is located in the Applications folder, stored under Utilities. Figure 2.11 shows its icon.

Figure 2.11 Terminal program icon

Start the Terminal application. You'll see a window that looks like Figure 2.12.

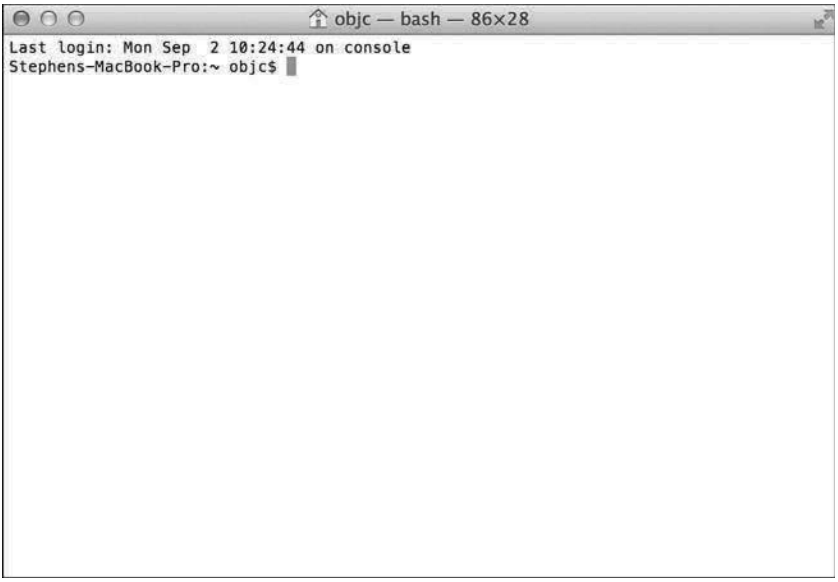

Figure 2.12 Terminal window

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 that 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) and have verified that you have the right tools installed, 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 files -o program
```

files is the list of files to be compiled. In this example, we have only one such file, and we're calling it `main.m`. *program* 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 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
2012-09-03 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

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 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:

```
// First program example

#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/Fraction.pdf
*/
```

Which style of comment you use is entirely up to you. Just note that you cannot 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 much 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 can also 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 as follows:

```
@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, "Numbers, Strings, and Collections."

The `NSLog` routine is a function 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 not described 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.

Now that you 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 `int`eger. 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` displays immediately following the phrase "The sum of." Similarly, the values of `value2` and `sum` will print 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 about 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;
}
```

Index

Symbols

B

C

G

H

I

N

O

Q

R

V

W

X-Y-Z