Learn Objective-C on the Mac

Penciled by MARK DALRYMPLE
Inked by SCOTT KNASTER

Apress®

Learn Objective-C on the Mac

ISBN-13 (pbk): 978-1-4302-1815-9

ISBN-13 (electronic): 978-1-4302-1816-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Clay Andres and Dave Mark
Technical Reviewer: Jeff LaMarche
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Denise Santoro Lincoln
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor/Artist/Interior Designer: Diana Van Winkle
Proofreader: Greg Teague
Indexer: Toma Mulligan
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

For Jerri Shertzer—teacher, mentor, friend
—Mark

Contents at a Glance

Contents

About the Authors

Mark Dalrymple is a longtime Mac and Unix programmer who has worked on cross-platform toolkits, Internet publishing tools, high-performance web servers, and end-user desktop applications. He's also the principal author of *Advanced Mac OS X Programming* (Big Nerd Ranch 2005). In his spare time, he plays trombone and bassoon and makes balloon animals.

Scott Knaster is a legendary (that is, very old) Mac programmer and author of such best-selling books as *Take Control of Switching to the Mac* (TidBITS Publishing Inc. 2008) and *Macintosh Programming Secrets* (Addison-Wesley 1992). His book *How to Write Macintosh Software* (Addison-Wesley 1992) was required reading for Mac programmers for more than a decade. He lives in a house with other people and a dog.

About the Technical Reviewer

Jeff LaMarche is a longtime Mac developer and certified Apple iPhone developer with more than 20 years of programming experience. He's written on Cocoa and Objective-C for *MacTech Magazine*, as well as articles for Apple's Developer Technical Services web site. He has experience working in enterprise software as both a developer for PeopleSoft, starting in the late 1990s, and later as an independent consultant.

Acknowledgments

If you've ever read a technical book, you've seen the acknowledgments and understand that even though there are (in this case) two names on the front cover, a lot of other folks behind the scenes make the whole process work.

In particular, we'd like to single out Denise Santoro Lincoln, who was our primary wrangler. We gave her "polenta" of problems, which she handled with taste, grace, and humor. Thanks also to Clay Andres and Jeff LaMarche, who helped make sure we didn't tell you any lies. Zillions of thanks to Laura Esterman, our production editor, for turning mere piles of text into this awesome tome that you're reading and to Heather Lang for warping (temporarily, we hope) her mind sufficiently to think like we do and still perform a masterful copy editing job.

Mark would like to thank Aaron Hillegass for introducing him to all of this Objective-C and Cocoa stuff many moons ago and for introducing him to Scott and Dave. Without Aaron, none of this would have happened. Also, Mark gives a shout out to Greg Miller for introducing him to the coolness of KVC and `NSPredicate`. And Scott just wants to thank Mark for doing all the real work.

Finally, impossibly enormous thanks go out to Dave Mark. Without his vision, dogged persistence, and awesome nagging, this book would not have seen the light of day.

Preface

One of the dangers of being a programmer for a long time is that you can lose that spark of delight that got you interested in programming the first place. Luckily, shiny new technologies come along all the time that can reignite that interest, and Mac OS X is chock full of shiny stuff.

Objective-C is a programming language that blends C's speed and ubiquity with an elegant object-oriented environment and provides a buzzword-laden cornucopia of programming good times. Objective-C is the gateway drug for many of Apple's niftiest technologies, such as the Cocoa toolkit and the iPhone SDK. Once you've mastered the Objective-C language, you're well on your way to conquering the rest of the platform. And from there, you can try to take over the world.

Chapter 1

Hello

Welcome to *Learn Objective-C on the Mac*! This book is designed to teach you the basics of the Objective-C language. Objective-C is a superset of C and is the language used by many (if not most) applications that have a true Mac OS X look and feel.

This book teaches you the Objective-C language and introduces you to its companion, Apple's Cocoa toolkit. Cocoa is written in Objective-C and contains all the elements of the Mac OS X user interface, plus a whole lot more. Once you learn Objective-C in this book, you'll be ready to dive into Cocoa with a full-blown project or another book such as *Learn Cocoa on the Mac* or *Beginning iPhone Development*, both by Dave Mark and Jeff LaMarche (Apress 2009).

In this chapter, we'll let you know the basic information you need before you get started with this book. We'll also serve up a bit of history about Objective-C and give you a thumbnail sketch of what's to come in future chapters.

Before You Start

Before you read this book, you should have some experience with a C-like programming language such as C++, Java, or venerable C itself. Whatever the language, you should feel comfortable with its basic principles. You should know what variables and functions are and understand how to control your program's flow using conditionals and loops. Our focus is the features Objective-C adds to its base language, C, along with some goodies chosen from Apple's Cocoa toolkit.

Are you coming to Objective-C from a non-C language? You'll still be able to follow along, but you might want to take a look at Appendix A or check out *Learn C on the Mac* by Dave Mark (Apress 2009).

Where the Future Was Made Yesterday

Cocoa and Objective-C are at the heart of Apple's Mac OS X operating system. Although Mac OS X is relatively new, Objective-C and Cocoa are much older. Brad Cox invented Objective-C in the early 1980s to meld the popular and portable C language with the elegant Smalltalk language. In 1985, Steve Jobs founded NeXT, Inc., to create powerful, affordable workstations. NeXT chose Unix as its operating system and created NextSTEP, a powerful user interface toolkit developed in Objective-C. Despite its features and a small, loyal following, NextSTEP achieved little commercial success.

When Apple acquired NeXT in 1996 (or was it the other way around?), NextSTEP was renamed Cocoa and brought to the wider audience of Macintosh programmers. Apple gives away its development tools—including Cocoa—for free, so any Mac programmer can take advantage of them. All you need is a bit of programming experience, basic knowledge of Objective-C, and the desire to dig in and learn stuff.

You might wonder, "If Objective-C and Cocoa were invented in the '80s—in the days of *Alf* and *The A-Team*, not to mention stuffy old Unix—aren't they old and moldy by now?" Absolutely not! Objective-C and Cocoa are the result of years of effort by a team of excellent programmers, and they have been continually updated and enhanced. Over time, Objective-C and Cocoa have evolved into an incredibly elegant and powerful set of tools. Objective-C is also the key to writing applications for the iPhone. So now, twenty-some years after NeXT adopted Objective-C, all the cool kids are using it.

What's Coming Up

Objective-C is a superset of C. Objective-C begins with C, and then adds a couple of small but significant additions to the language. If you've ever looked at C++ or Java, you may be surprised at how small Objective-C really is. We'll cover Objective-C's additions to C in detail in this book's chapters:

- Chapter 2, "Extensions to C," focuses on the basic features that Objective-C introduces.
- In Chapter 3, "An Introduction to Object-Oriented Programming," we kick off the learning by showing you the basics of object-oriented programming.
- Chapter 4, "Inheritance," describes how to create classes that gain the features of their parent classes.
- Chapter 5, "Composition," discusses techniques for combining objects so they can work together.
- Chapter 6, "Source File Organization," presents real-world strategies for creating your program's sources.

- Chapter 7, "More about Xcode," shows you some shortcuts and power-user features to help you get the most out of your programming day.
- We take a brief respite from Objective-C in Chapter 8, "A Quick Tour of the Foundation Kit," to impress you with some of Cocoa's cool features using one of its two primary frameworks.
- You'll spend a lot of time in your Cocoa applications dealing in Chapter 9, "Memory Management" (sorry about that).
- Chapter 10, "Object Initialization," is all about what happens when objects are born.
- Chapter 11, "Properties," gives you the lowdown on Objective-C's new dot notation and an easier way to make object accessors.
- Chapter 12, "Categories," describes the supercool Objective-C feature that lets you add your own methods to existing classes—even those you didn't write.
- Chapter 13, "Protocols," tells about a form of inheritance in Objective-C that allows classes to implement packaged sets of features.
- Chapter 14, "Introduction to the Application Kit," gives you a taste of the gorgeous applications you can develop in Cocoa using its other primary framework.
- Chapter 15, "File Loading and Saving," shows you how to save and retrieve your data.
- Chapter 16, "Key-Value Coding," gives you ways to deal with your data indirectly.
- And finally, in Chapter 17, "NSPredicate," we show you how to slice and dice your data.

If you're coming from another language like Java or C++, or from another platform like Windows or Linux, you may want to check out Appendix A, "Coming to Objective-C from Other Languages," which points out some of the mental hurdles you'll need to jump to embrace Objective-C.

Summary

Mac OS X programs are written in Objective-C, using technology from way back in the 1980s that has matured into a powerful set of tools. In this book, we'll start by assuming you know something about C programming and go from there.

We hope you enjoy the ride!

Chapter 2

Extensions to C

Objective-C is nothing more than the C language with some extra features drizzled on top—it's delicious! In this chapter, we'll cover some of those key extras as we take you through building your first Objective-C program.

The Simplest Objective-C Program

You've probably seen the C version of the classic Hello World program, which prints out the text "Hello, world!" or a similar pithy remark. Hello World is usually the first program that neophyte C programmers learn. We don't want to buck tradition, so we're going to write a similar program here called Hello Objective-C.

Building Hello Objective-C

As you work through this book, we're assuming you have Apple's Xcode tools installed. If you don't already have Xcode, or if you've never used it before, an excellent section in Chapter 2 of Dave Mark's *Learn C on the Mac* (Apress 2009) walks you through the steps of acquiring, installing, and creating programs with Xcode.

In this section, we'll step through the process of using Xcode to create your first Objective-C project. If you are already familiar with Xcode, feel free to skip ahead; you won't hurt our feelings. Before you go, be sure to expand the *Learn ObjC Projects* archive from this book's archive (which you can download from the Source Code/Download page of the Apress web site). This project is located in the *02.01 - Hello Objective-C* folder.

To create the project, start by launching Xcode. You can find the Xcode application in */Developer/Applications*. We put the Xcode icon in the Dock for easy access. You might want to do that too.

Once Xcode finishes launching, choose **New Project** from the **File** menu. Xcode shows you a list of the various kinds of projects it can create. Use your focus to ignore most of the intriguing project types there, and choose *Command Line Utility* on the left-hand side of the window and *Foundation Tool* on the right-hand side, as shown in Figure 2-1. Click the *Choose* button.

Figure 2-1. *Making a new foundation tool*

Xcode drops a sheet and asks you to name the project. You can choose any name you want, but as you can see in Figure 2-2, we called it Hello Objective-C. We're putting it into one of our *Projects* directories here to keep things organized, but you can put it anywhere you want.

After you click *Save*, Xcode shows you its main window, called the project window (see Figure 2-3). This window displays the pieces that compose your project along with an editing pane. The highlighted file, *Hello Objective-C.m*, is the source file that contains the code for Hello Objective-C.

Hello Objective-C.m contains boilerplate code, kindly provided by Xcode for each new project. We can make our Hello Objective-C application a little simpler than the sample Xcode supplies. Delete everything in *Hello Objective-C.m* and replace it with this code:

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

int main (int argc, const char *argv[])
{
  NSLog (@"Hello, Objective-C!");

  return (0);

} // main
```

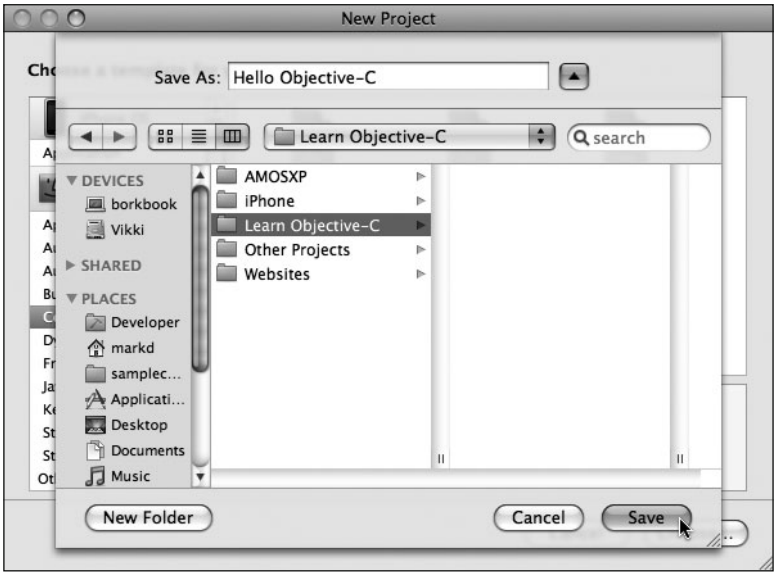


Figure 2-2. *Name the new foundation tool*

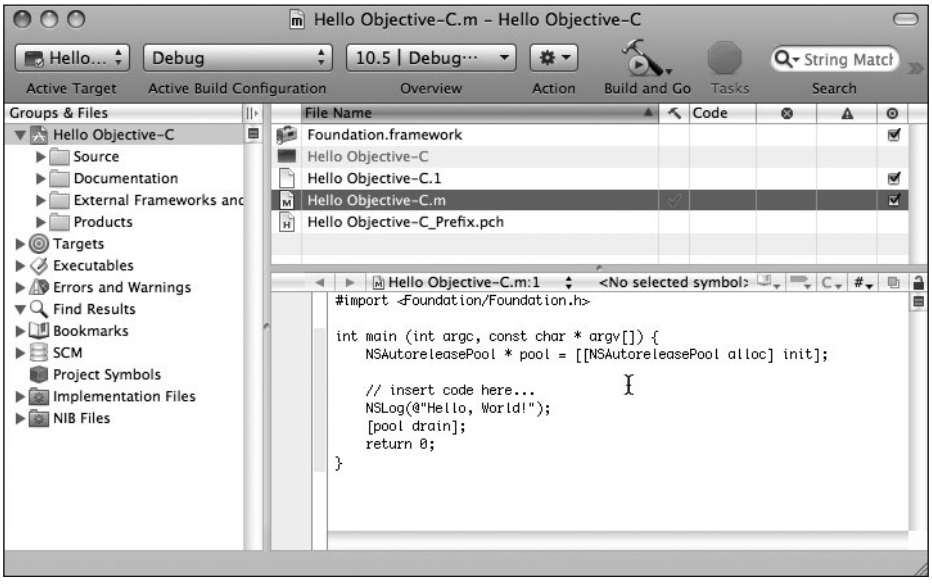


Figure 2-3. *XCode's main window*

If you don't understand all the code right now, don't worry about it. We'll go through this program in excruciating, line-by-line detail soon.

Source code is no fun if you can't turn it into a running program. Build and run the program by clicking the *Build and Go* button or pressing ⌘**R**. If there aren't any nasty syntax errors, Xcode compiles and links your program and then runs it. Open the Xcode console window

(by selecting **Console** from the **Run** menu or pressing ⌘⇧**R**), which displays your program's output, as shown in Figure 2-4.

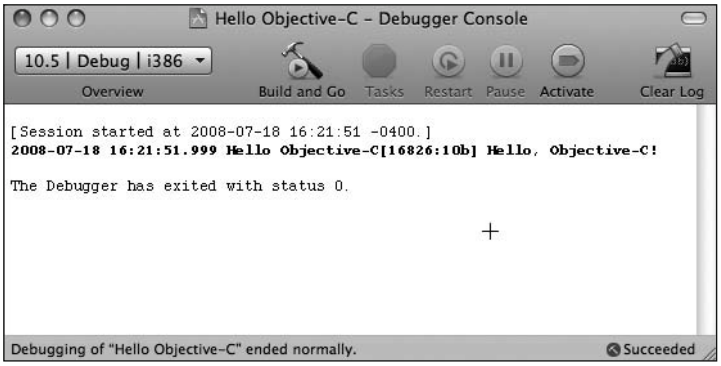


Figure 2-4. *Running Hello Objective-C*

And there you have it: your first working Objective-C program. Congratulations! Let's pull it apart and see how it works.

Deconstructing Hello Objective-C

Here, again, are the contents of *Hello Objective-C.m*:

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

int main (int argc, const char *argv[])
{
    NSLog (@"Hello, Objective-C!");

    return (0);

} // main
```

Xcode uses the *.m* extension to indicate a file that holds Objective-C code and will be processed by the Objective-C compiler. File names ending in *.c* are handled by the C compiler, and *.cpp* files are the province of the C++ compiler. (In Xcode, all this compiling is handled by the GNU Compiler Collection [GCC], a single compiler that understands all three variations of the language.)

The *main.m* file contains two lines of code that should be familiar to you already if you know plain C: the declaration of `main()` and the `return (0)` statement at the end. Remember that Objective-C really is C at heart, and the syntax for declaring `main()` and returning a value is the same as in C. The rest of the code looks slightly different from regular C. For example, what is that wacky `#import` thing? To find out, read on!

NOTE

The *.m* extension originally stood for "messages" when Objective-C was first introduced, referring to a central feature of Objective-C that we'll talk about in future chapters. Nowadays, we just call them dot-m files.

That Wacky #import Thing

Just like C, Objective-C uses **header files** to hold the declarations of elements such as structs, symbolic constants, and function prototypes. In C, you use the `#include` statement to inform the compiler that it should consult a header file for some definitions. You can use `#include` in Objective-C programs for the same purpose, but you probably never will. Instead, you'll use `#import`, like this:

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

`#import` is a feature provided by the GCC compiler, which is what Xcode uses when you're compiling Objective-C, C, and C++ programs. `#import` guarantees that a header file will be included only once, no matter how many times the `#import` directive is actually seen for that file.

NOTE

In C, programmers typically use a scheme based on the `#ifdef` directive to avoid the situation where one file includes a second file, which then, recursively, includes the first.

In Objective-C, programmers use `#import` to accomplish the same thing.

The `#import <Foundation/Foundation.h>` statement tells the compiler to look at the `Foundation.h` header file in the Foundation framework.

What's a framework? We're glad you asked. A framework is a collection of parts—header files, libraries, images, sounds, and more—collected together into a single unit. Apple ships technologies such as Cocoa, Carbon, QuickTime, and OpenGL as sets of frameworks. Cocoa consists of a pair of frameworks, Foundation and Application Kit (also known as AppKit), along with a suite of supporting frameworks, including Core Animation and Core Image, which add all sorts of cool stuff to Cocoa.

The Foundation framework handles features found in the layers beneath the user interface, such as data structures and communication mechanisms. All the programs in this book are based on the Foundation framework.

NOTE

Once you finish this book, your next step along the road to becoming a Cocoa guru is to master Cocoa's Application Kit, which contains Cocoa's high-level features: user interface elements, printing, color and sound management, AppleScript support, and so on. To find out more, check out *Learn Cocoa on the Mac* by Dave Mark and Jeff LaMarche (Apress 2009).

Each framework is a significant collection of technology, often containing dozens or even hundreds of header files. Each framework has a master header file that includes all the framework's individual header files. By using `#import` on the master header file, you have access to all the framework's features.

The header files for the Foundation framework take up nearly a megabyte of disk storage, and contain more than 14,000 lines of code, spread across over a hundred files. When you include the master header file with `#import <Foundation/Foundation.h>`, you get that whole vast collection. You might think wading through all that text for every file would take the compiler a lot of time, but Xcode is smart: it speeds up the task by using precompiled headers, a compressed and digested form of the header that's loaded quickly when you `#import` it.

If you're curious about which headers are included with the Foundation framework, you can peek inside its *Headers* directory (*/System/Library/Frameworks/Foundation.framework/Headers/*). You won't break anything if you browse the files in there; just don't remove or change anything.

NSLog() and @"strings"

Now that we have used `#import` on the master header file for the Foundation framework, you're ready to write code that takes advantage of some Cocoa features. The first (and only) real line of code in Hello Objective-C uses the `NSLog()` function, like so:

```
NSLog (@"Hello, Objective-C!");
```

This prints "Hello, Objective-C!" to the console. If you've used C at all, you have undoubtedly encountered `printf()` in your travels. `NSLog()` is a Cocoa function that works very much like `printf()`.

Just like `printf()`, `NSLog()` takes a string as its first argument. This string can contain format specifiers (such as `%d`), and the function takes additional parameters that match the format specifiers. `printf()` plugs these extra parameters into the string before it gets printed.

As we've said before, Objective-C is just C with a little bit of special sauce, so you're welcome to use `printf()` instead of `NSLog()` if you want. We recommend `NSLog()`, however, because it adds features such as time and date stamps, as well as automatically appending the newline (`'\n'`) character for you.

You might be thinking that `NSLog()` is kind of a strange name for a function. What is that "NS" doing there? It turns out that Cocoa prefixes all its function, constant, and type names with "NS". This prefix tells you the function comes from Cocoa instead of some other toolkit.

The prefix helps prevent **name collisions**, big problems that result when the same identifier is used for two different things. If Cocoa had named this function `Log()`, there's a good chance the name would clash with a `Log()` function created by some innocent programmer somewhere. When a program containing `Log()` is built with Cocoa included, Xcode complains that `Log()` is defined multiple times, and sadness results.

Now that you have an idea why a prefix is a good idea, you might wonder about the specific choice: why "NS" instead of "Cocoa," for example? Well, the "NS" prefix dates back from the time when the toolkit was called NextSTEP and was the product of NeXT Software (formerly NeXT, Inc.), which was acquired by Apple in 1996. Rather than break compatibility with code already written for NextSTEP, Apple just continued to use the "NS" prefix. It's a historical curiosity now, like your appendix.

Cocoa has staked its claim on the NS prefix, so obviously, you should not prefix any of your own variables or function names with "NS". If you do, you will confuse the readers of your code, making them think your stuff actually belongs to Cocoa. Also, your code might break in the future if Apple happens to add a function to Cocoa with the same name as yours. There is no centralized prefix registry, so you can pick your own prefix. Many people prefix names with their initials or company names. To make our examples a little simpler, we won't use a prefix for the code in this book.

Let's take another look at that `NSLog()` statement:

```
NSLog (@"Hello, Objective-C!");
```

Did you notice the at sign before the string? It's not a typo that made it past our vigilant editors. The at sign is one of the features that Objective-C adds to standard C. A string in double quotes preceded by an at sign means that the quoted string should be treated as a Cocoa `NSString` element.

So what's an `NSString` element? Peel the "NS" prefix off the name and you see a familiar term: "String". You already know that a string is a sequence of characters, usually human-readable, so you can probably guess (correctly) that an `NSString` is a sequence of characters in Cocoa.

`NSString` elements have a huge number of features packed into them and are used by Cocoa any time a string is needed. Here are just a few of the things an `NSString` can do:

- Tell you its length
- Compare itself to another string
- Convert itself to an integer or floating-point value

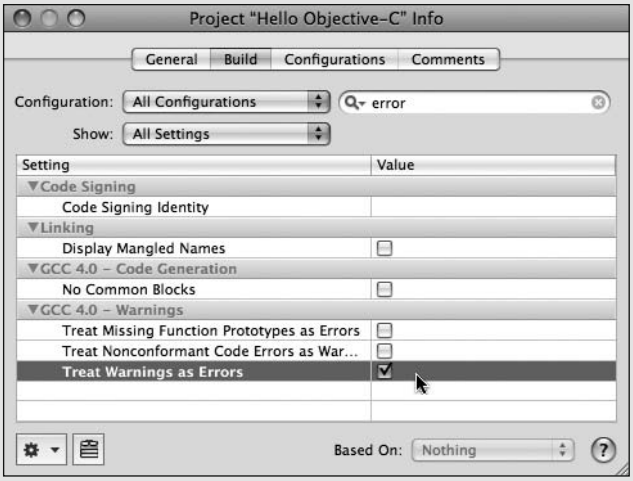
That's a whole lot more than you can do with C-style strings. We'll be using and exploring `NSString` elements much more in Chapter 8.

WATCH THOSE STRINGS

One mistake that's easy to make is to pass a C-style string to `NSLog()` instead of one of the fancy `NSString @"strings"` elements. If you do this, the compiler will give you a warning:

```
main.m:46: warning: passing arg 1 of `NSLog' from
      incompatible pointer type
```

If you run this program, it might crash. To catch problems like this, you can tell Xcode to always treat warnings as errors. To do that, select the top item in the Xcode *Groups & Files* list, choose File ➤ Get Info, select the *Build* tab, type *error* into the search field, and check the *Treat Warnings as Errors* checkbox, as shown in the following image. Also make sure that the *Configuration* pop-up menu at the top says *All Configurations*.



Here's another cool fact about `NSString`: the name itself highlights one of the nice features of Cocoa. Most Cocoa elements are named in a very straightforward manner, striving to describe the features they implement. For instance, `NSArray` provides arrays; `NSDateFormatter` helps you format dates in different ways; `NSThread` gives you tools for multithreaded programming; and `NSSpeechSynthesizer` lets you hear speech.

Now, we'll get back to stepping through our little program. The last line of the program is the `return` statement that ends the execution of `main()` and finishes the program:

```
return (0);
```

The zero value returned says that our program completed successfully. This is just the way `return` statements work in C.

Congratulations, again! You've just written, compiled, run, and dissected your first Objective-C program.

Are You the Boolean Type?

Many languages have a Boolean type, which is, of course, a fancy term for variables that store true and false values. Objective-C is no exception.

C has a Boolean data type, `bool`, which can take on the values `true` and `false`. Objective-C provides a similar type, `BOOL`, which can have the values `YES` and `NO`. Objective-C's `BOOL` type, incidentally, predates C's `bool` type by over a decade. The two different Boolean types can coexist in the same program, but when you're writing Cocoa code, you'll be using `BOOL`.

NOTE

`BOOL` in Objective-C is actually just a type definition (`typedef`) for the signed character type (`signed char`), which uses 8 bits of storage. `YES` is defined as `1` and `NO` as `0` (using `#define`).

Objective-C doesn't treat `BOOL` as a true Boolean type that can hold only `YES` or `NO` values. The compiler considers `BOOL` to be an 8-bit number, and the values of `YES` and `NO` are just a convention. This causes a subtle gotcha: if you inadvertently assign an integer value that's more than 1 byte long, such as a `short` or an `int` value, to a `BOOL` variable, only the lowest byte is used for the value of the `BOOL`. If that byte happens to be zero (as with `8960`, which in hexadecimal is `0x2300`), the `BOOL` value will be zero, the `NO` value.

Mighty BOOL in Action

To show mighty BOOL in action, we move on to our next project, *02.02 - BOOL Party*, which compares pairs of integers to see if they're different. Aside from `main()`, the program defines two functions. The first, `areIntsDifferent()`, takes two integer values and returns a BOOL: YES if the integers are different and NO if they are the same. A second function, `boolString()`, takes a BOOL parameter and returns the string `@"YES"` if the parameter is YES and `@"NO"` if the parameter is NO. This is a handy function to have around when you want to print out a human-readable representation of BOOL values. `main()` uses these two functions to compare integers and print out the results.

Creating the project for BOOL Party is exactly the same process as making the project for Hello Objective-C:

1. Launch Xcode, if it's not already running.
2. Select **New Project** from the **File** menu.
3. Choose *Command Line Utility* on the left and *Foundation Tool* on the right.
4. Click *Choose*.
5. Type *BOOL Party* as the *Project Name*, and click *Save*.

Edit *BOOL Party.m* to make it look like this:

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

// returns NO if the two integers have the same
// value, YES otherwise

BOOL areIntsDifferent (int thing1, int thing2)
{
  if (thing1 == thing2) {
    return (NO);
  } else {
    return (YES);
  }

} // areIntsDifferent


// given a YES value, return the human-readable
// string "YES". Otherwise return "NO"

NSString *boolString (BOOL yesNo)
{
  if (yesNo == NO) {
    return (@"NO");
```

```
  } else {
    return (@"YES");
  }

} // boolString


int main (int argc, const char *argv[])
{
  BOOL areTheyDifferent;

  areTheyDifferent = areIntsDifferent (5, 5);

  NSLog (@"are %d and %d different? %@",
      5, 5, boolString(areTheyDifferent));

  areTheyDifferent = areIntsDifferent (23, 42);

  NSLog (@"are %d and %d different? %@",
      23, 42, boolString(areTheyDifferent));

  return (0);

} // main
```

Build and run your program. You'll need to bring up the *Console* window to see the output, by choosing **Console** from the **Run** menu, or using the keyboard shortcut ⌘⇧**R**. In the *Run Debugger Console* window, you should see output like the following:

```
2008-07-20 16:47:09.528 02 BOOL Party[16991:10b] are 5 and 5 different? NO
2008-07-20 16:47:09.542 02 BOOL Party[16991:10b] are 23 and 42 different?
YES

The Debugger has exited with status 0.
```

Once again, let's pull this program apart, function by function, and see what's going on. The first function in our tour is `areIntsDifferent()`:

```
BOOL areIntsDifferent (int thing1, int thing2)
{
  if (thing1 == thing2) {
    return (NO);
  } else {
    return (YES);
  }

} // areIntsDifferent
```

The areIntsDifferent() function that takes two integer parameters and returns a BOOL value. The syntax should be familiar to you from your C experience. Here you can see thing1 being compared to thing2. If they're the same, NO is returned (since they're not different). If they're different, YES is returned. That's pretty straightforward, isn't it?

WON'T GET BOOLED AGAIN

Experienced C programmers might be tempted to write the areIntsDifferent() function as a single statement:

```
BOOL areIntsDifferent_faulty (int thing1, int thing2)
{
  return (thing1 - thing2);
} // areIntsDifferent_faulty
```

They'd do so operating under the assumption that a nonzero value is the same as YES. But that's not the case. Yes, this function returns a value, as far as C is concerned, that is true or false, but callers of functions returning BOOL will expect either YES or NO to be returned. If a programmer tries to use this function as follows, it will fail, since 23 minus 5 is 18:

```
if (areIntsDifferent_faulty(23, 5) == YES) {
  // ....
}
```

While the preceding function may be a true value in C, it is not equal to YES (a value of 1) in Objective-C.

It's a good idea never to compare a BOOL value directly to YES, because too-clever programmers sometimes pull stunts similar to areIntsDifferent_faulty(). Instead, write the preceding if statement like this:

```
if (areIntsDifferent_faulty(5, 23)) {
  // ....
}
```

Comparing directly to NO is always safe, since falsehood in C has a single value: zero.

The second function, boolString(), maps a numeric BOOL value to a string that's readable by mere humans:

```
NSString *boolString (BOOL yesNo)
{
  if (yesNo == NO) {
    return (@"NO");
  } else {
    return (@"YES");
  }

} // boolString
```